

## 版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。



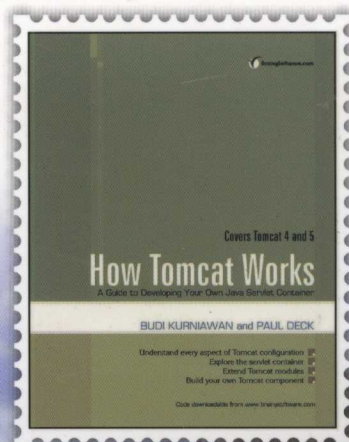
深入理解Tomcat的必读之作  
欲知其然，更欲知其所以然

# 深入剖析

# Tomcat

How Tomcat Works  
A Guide to Developing Your Own Java  
Servlet Container

(美) Budi Kurniawan 著  
Paul Deck 译  
曹旭东 译



机械工业出版社  
China Machine Press

## 本书特色

本书以Tomcat 4和Tomcat 5两个版本为基础，从建立一个最简单的连接开始，深入介绍Tomcat的体系结构。从连接器到最终的JMX管理，循序渐进，层层深入。每一章有配有相关代码，既是对理论内容的具体展现，也可以帮助读者编写一个实用的应用服务器。

在内容上，本书更关注对Tomcat基本体系结构的讲解，并没有涉及在实际应用中的具体实现细节。希望读者在阅读的时候，着重把握Tomcat的一些设计思想，在此基础上，再针对某一方面进行深入的学习和研究。

华章专业开发者丛书

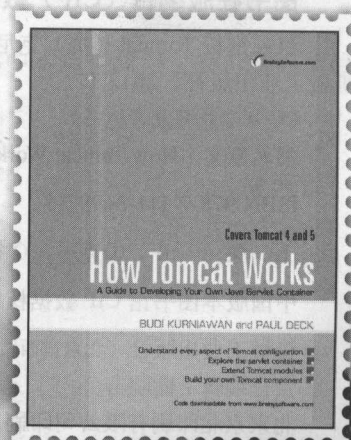
# 深入剖析

# Tomcat

How Tomcat Works  
A Guide to Developing Your Own Java  
Servlet Container

(美) Budi Kurniawan 著  
Paul Deck

曹旭东 译



机械工业出版社  
China Machine Press



本书深入剖析 Tomcat 4 和 Tomcat 5 中的每个组件,并揭示其内部工作原理。通过学习本书,你将可以自行开发 Tomcat 组件,或者扩展已有的组件。Tomcat 是目前比较流行的 Web 服务器之一。作为一个开源和小型的轻量级应用服务器, Tomcat 易于使用,便于部署,但 Tomcat 本身是一个非常复杂的系统,包含了很多功能模块。这些功能模块构成了 Tomcat 的核心结构。本书从最基本的 HTTP 请求开始,直至使用 JMX 技术管理 Tomcat 中的应用程序,逐一剖析 Tomcat 的基本功能模块,并配以示例代码,使读者可以逐步实现自己的 Web 服务器。

Authorized translation from the English language edition entitled How Tomcat Works: A Guide to Developing Your Own Java Servlet Container by Budi Kurniawan and Paul Deck, published by Brainy Software, Inc., Copyright © 2004.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without permission of Brainy Software, Inc.

Chinese simplified language edition published by China Machine Press.

Copyright © 2012 by China Machine Press.

本书中文简体字版由 Brainy Software 授权机械工业出版社独家出版。未经出版者书面许可,不得以任何方式复制或抄袭本书内容。

封底无防伪标均为盗版

版权所有,侵权必究

本书法律顾问 北京市展达律师事务所

本书版权登记号:图字:01-2011-7874

图书在版编目(CIP)数据

深入剖析 Tomcat / (美) 克尼亚万(Kurniawan, B.), (美) 德克(Deck, P.) 著; 曹旭东译. —北京:机械工业出版社, 2012.1

(华章专业开发者丛书)

书名原文: How Tomcat Works: A Guide to Developing Your Own Java Servlet Container

ISBN 978-7-111-36997-4

I. 深… II. ①克… ②德… ③曹… III. JAVA 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字(2011)第 277355 号

机械工业出版社(北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑:谢晓芳

北京京北印刷有限公司印刷

2012 年 5 月第 1 版第 2 次印刷

186mm×240mm·22.25 印张

标准书号: ISBN 978-7-111-36997-4

定价: 59.00 元

凡购本书,如有缺页、倒页、脱页,由本社发行部调换

客服热线:(010) 88378991; 88361066

购书热线:(010) 68326294; 88376949; 68995259

投稿热线:(010) 88379604

读者信箱:hzjsj@hzbook.com

# 译者序

Tomcat 是 Apache 软件基金会 (Apache Software Foundation) 的一个顶级项目, 由 Apache、Sun 和其他一些公司及个人共同开发, 是目前比较流行的 Web 服务器之一。作为一个开源的、小型的轻量级应用服务器, Tomcat 深受广大程序员的喜爱, 具有占用系统资源少, 扩展性好, 支持负载均衡与邮件服务等开发应用系统常用的功能; 而且它还在不断地改进和完善中, 任何一个感兴趣的程序员都可以更改它或在其中加入新的功能。

虽然 Tomcat 易于使用, 便于部署, 但 Tomcat 本身是一个非常复杂的系统, 包含了很多功能模块。这些功能模块密切合作, 各司其职, 构成了 Tomcat 的核心结构。作者从最基本的 HTTP 请求开始, 直至使用 JMX 技术管理 Tomcat 中的应用程序, 逐步深入, 逐一剖析 Tomcat 的基本功能模块, 并配以示例代码, 使读者可以逐步实现自己的一个 Web 服务器。

当然, 本书并不能完全覆盖 Tomcat, 书中并没有包含 Tomcat 的太多设计思路及具体的实现细节, 而是更注重对 Tomcat 基本结构的分析介绍。在每一章中都有与本章内容相关的示例应用程序, 帮助读者更好地理解该章的内容。

本书由曹旭东翻译, 由于时间仓促, 加上译者水平有限, 书中难免有疏漏之处, 望广大读者予以指正。

曹旭东

# 前言

欢迎阅读本书。本书剖析了 Tomcat 4.1.12 版本和 Tomcat 5.0.18 版本的基本结构，并解释了其 servlet 容器 Catalina 的内部工作原理。Catalina 是开源、免费的，也是最受欢迎的 servlet 容器之一。Tomcat 本身是一个复杂的系统，包含了许多不同的组件。若你想学习 Tomcat 的工作方式，应该从了解这些组件开始。本书描述 Tomcat 的总体结构，并针对每个组件建立一个简单的版本，使你更好地理解组件的工作机制，之后对真实组件进行描述。

“本书结构”一节会对全书的章节设置做一个总体介绍，并说明构建的应用程序的总体结构。在“准备必需的软件”一节，说明需要下载使用哪些软件，如何为代码创建目录结构等。

## 本书读者对象

本书适合于所有使用 Java 技术工作的开发人员。

- 如果你是一名 JSP/Servlet 程序员或 Tomcat 用户，并想了解 servlet 容器是如何工作的，那么本书很适合你；
- 如果你想加入 Tomcat 开发团队，那么本书很适合你，因为你首先要学习已有的代码是如何工作的；
- 如果你不是一名 Web 开发人员，但对软件开发很有兴趣，那么你可以从本书中学习到一个大型应用软件（如 Tomcat）是如何设计和开发的；
- 如果你想对 Tomcat 进行配置或定制，你应该阅读本书。

为了更好地理解本书所讲述的内容，你需要理解 Java 中的面向对象编程知识，以及 Servlet 编程方面的知识。如果你对后者还不熟悉，那么你学习起来可能会有些困难。你可以先学习一下 Servlet 编程方面的知识，例如看一下 Budi 的《Java for the Web with Servlets, JSP, and EJB》一书。为了使你更好地理解本书的内容，每一章的开头都会有一段与该章内容相关的背景信息的介绍。

## servlet 容器是如何工作的

servlet 容器是一个复杂的系统，但是，它有 3 个基本任务，对每个请求，servlet 容器会为其完成以下 3 个操作：

- 创建一个 request 对象，用可能会在调用的 Servlet 中使用到的信息填充该 request 对象，如参数、头、cookie、查询字符串、URI 等。request 对象是 `javax.servlet.ServletRequest` 接口或 `javax.servlet.http.HttpServletRequest` 接口的一个实例。
- 创建一个调用 Servlet 的 response 对象，用来向 Web 客户端发送响应。response 对象是 `javax.servlet.ServletResponse` 接口或 `javax.servlet.http.HttpServletResponse` 接口的一个实例；



- 调用 Servlet 的 service() 方法，将 request 对象和 response 对象作为参数传入。Servlet 从 request 对象中读取信息，并通过 response 对象发送响应信息。

当你阅读具体的章节时，你会看到关于 servlet 容器 Catalina 的详细描述。

## Catalina 框图

Catalina 是一个成熟的软件，设计和开发得十分优雅，功能结构也是模块化的。上一节“servlet 容器是如何工作的”中提到了 servlet 容器的任务，基于这些任务可以将 Catalina 划分为两个模块：连接器（connector）和容器（container）。

图 I-1 很简单，在后续的章节中，你会逐个接触到所有的组件。

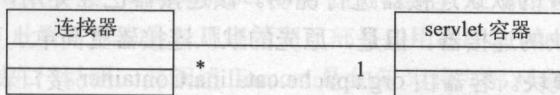


图 I-1 Catalina 的主要模块

现在，回到图 I-1，这里的连接器负责将一个请求与容器相关联。它的工作包括为它接收到的每个 HTTP 请求创建一个 request 对象和一个 response 对象。然后，它将处理过程交给容器。容器从连接器中接收到 request 对象和 response 对象，并负责调用相应的 Servlet 的 service() 方法。

但是请记住，上面所描述的处理过程只是 Catalina 容器处理请求的整个过程的一小部分，犹如冰山的一角，在容器中还包括很多其他的事情要做。例如，在容器调用相应的 Servlet 的 service() 方法之前，它必须先载入该 Servlet 类，对用户进行身份验证（如果有必要的话），为用户更新会话信息等。因此，当你发现容器使用了很多不同的模块来处理这些事情时，请不要太惊讶。例如，管理器模块用来处理用户会话信息，载入器模块用来载入所需的 Servlet 类等。

## Tomcat 4 和 Tomcat 5

本书涵盖了 Tomcat 4 和 Tomcat 5 两个版本。下面是这两个版本的一些区别之处：

- Tomcat 5 支持 Servlet 2.4 和 JSP 2.0 规范，Tomcat 4 支持 Servlet 2.3 和 JSP 1.2 规范；
- Tomcat 5 默认的连接器的比 Tomcat 4 默认的连接器的执行效率更高；
- Tomcat 5 使用共享线程来执行后台任务，而 Tomcat 4 的组件使用各自的线程执行后台任务，因此，相比于 Tomcat 4，Tomcat 5 更节省资源；
- Tomcat 5 不再使用映射器组件来查找子组件，因此，代码更简单。

## 本书结构

本书共 20 章，前两章概述了全书内容。第 1 章介绍了 HTTP 服务器是如何工作的，第 2 章介绍了一个简单的 servlet 容器。第 3 章和第 4 章着重于连接器的说明，第 5~20 章介绍容器中的各个组件。下面是每一章的内容简介。

**注意** 每一章都配有一个应用程序用于对该章所介绍组件进行实际应用的说明。

第 1 章：本书从介绍一个简单的 HTTP 服务器开始。为了建立一个可以运行的 HTTP 服务器，你需要了解 java.net 包下 Socket 类和 ServerSocket 类的内部运行机制。该章有详细的背景信息介绍，使你可以理解该章中应用程序的运行机制。

第 2 章：阐明一个简单的 servlet 容器是如何工作的。该章有两个与 servlet 容器有关的应用程序，可以服务于静态资源的请求和简单 Servlet 的请求。此外，你会学习到如何创建 request 对象和 response 对象，并将它们传递给被请求的 Servlet 的 service() 方法。此外，在该 servlet 容器中有一个可以运行的 Servlet，可以从 Web 浏览器中进行调用。

第 3 章：将对 Tomcat 4 中的默认连接器的精简版进行说明。该章中建立的应用程序可以作为一个学习工具，有助于理解在第 4 章中讨论的连接器。

第 4 章：对 Tomcat 4 的默认连接器进行说明。该连接器已经弃用，而是推荐使用另一个称为 Coyote 的执行速度更快的连接器。但是，原先的默认连接器更简单、更容易理解。

第 5 章：讨论容器模块。容器由 org.apache.catalina.Container 接口表示，共有 4 种类型的容器，分别是 Engine、Host、Context 和 Wrapper。该章提供了两个分别与 Context 和 Wrapper 相关的应用程序。

第 6 章：对 Lifecycle 接口进行说明。该接口定义了 Catalina 组件的生命周期，并提供了一种优雅的方法来通知其他组件在该组件中发生了某种事件。此外，Lifecycle 接口提供了一种优雅的机制来启动和关闭 Catalina 中所有的组件，只需要启动 / 关闭一次即可。

第 7 章：介绍日志记录器组件，该组件用于记录错误消息和其他的相关消息。

第 8 章：对载入器组件进行介绍，载入器是 Catalina 中的重要模块，负责载入 Servlet 和 Web 应用程序中所需要的其他类。该章还将说明 Web 应用程序的重载是如何实现的。

第 9 章：介绍管理器组件。该组件负责在会话管理中管理会话。该章介绍了几种不同类型的管理器，并说明了管理器组件是如何持久化一个 session 对象的。在该章末尾，介绍如何使用 StandardManager 实例构建一个使用 session 对象保存数据的应用程序。

第 10 章：该章将讨论 Web 应用程序的安全限制，以限制对某些内容的访问。你会学习到一些与安全相关的实体，如主体、角色、登录配置和身份验证等。该章有两个应用程序，分别在 StandardContext 对象中安装了验证器阀，并使用基本验证来对用户进行身份验证。

第 11 章：对 org.apache.catalina.core.StandardWrapper 类进行了详细阐述，该类表示 Web 应用程序中的一个 Servlet 对象。该章还说明了过滤器和 Servlet 的 service() 方法是如何调用的。该章的应用程序使用 StandardWrapper 实例来表示实际的 Servlet 对象。

第 12 章：介绍 org.apache.catalina.core.StandardContext 类，该类表示一个 Web 应用程序。该章还说明了如何对一个 StandardContext 对象进行配置（这是在接收到 HTTP 请求时进行的）、如何支持 Web 应用程序的自动重载，以及 Tomcat 5 如何通过一个共享线程来执行其相关联组件中的周期性任务。

第 13 章：对另外两种容器（Host 和 Engine）进行说明。该章对这两种容器的标准实现 org.apache.catalina.core.StandardHost 和 org.apache.catalina.core.StandardEngine 进行了说明。

第 14 章：对服务器组件和服务组件进行介绍。服务器组件提供了一种优雅的机制来启动或关闭整个 servlet 容器，服务组件可作为一个容器和一个或多个连接器的持有者。该章通过应用



程序来说明如何使用服务器组件和服务组件。

第 15 章：说明如何通过 Digester 组件来对 Web 应用程序进行配置。Digester 是 Apache 软件基金会的一个开源项目。即使你对这个不熟悉也没有关系，该章会简要介绍 Digester 库，说明如何使用该库来将 XML 文档中的节点转换为 Java 对象。该章然后说明 Tomcat 是如何通过 ContextConfig 对象来对 StandardContext 对象进行配置的。

第 16 章：对 Tomcat 中的关闭钩子进行说明。不论用户如何关闭 Tomcat（即通过发送关闭命令，或是突然直接关闭控制台），通过使用关闭钩子，Tomcat 总是可以执行一些清理工作。

第 17 章：对使用批处理文件和 Shell 脚本来启动和关闭 Tomcat 进行说明。

第 18 章：对部署器组件进行说明，该组件负责部署和安装 Web 应用程序。

第 19 章：对一个特殊接口（ContainerServlet）进行说明。使用该接口，Servlet 对象可以访问 Catalina 中的内部对象。此外，该章会对用来管理已部署应用程序的 Manager 应用程序进行说明。

第 20 章：对 JMX 进行说明，并阐述 Tomcat 是如何为其内部对象创建 MBean，并使这些内部对象可托管的。

## 每一章的应用程序

每一章都会有一个或多个应用程序用来解释 Catalina 中的某个特定组件的使用方法。一般情况下，在该应用程序中你会找到该组件的精简版，或是为说明如何使用 Catalina 组件而编写的代码。在每一章的应用程序中编写的所有的类和接口都在 *ex[章号].pyrmont* 包下。例如，第 1 章的应用程序类会在 *ex01.pyrmont* 包下。

## 准备必需的软件

本书的应用程序会运行在 J2SE 的 1.4 版本下。源文件的压缩包可以从作者的网站 [www.brainysoftware.com](http://www.brainysoftware.com) 上下载。该压缩包包含了 Tomcat 4.1.12 的源代码，以及本书中的应用程序的代码。假设你已经安装了 J2SE 的 1.4 版本，而且环境变量 *path* 也已经包含了 JDK 的路径，那么只需执行下面的步骤。

1) 解压缩 zip 文件。解压缩后会有一个名为 HowTomcatWorks 的新文件夹。HowTomcatWorks 是工作目录，其下有一些子目录，包括 *lib*（包含所有必需的库文件）、*src*（包含所有源文件）、*webroot*（包含一个 HTML 文件和三个 Servlet 示例文件）和 *webapps*（包含示例应用程序）；

2) 进入到工作目录中，编译 Java 文件。若你使用 Windows 操作系统，则运行 *win-compile.bat* 批处理文件。若你使用 Linux 操作系统，则执行下面的命令（必要时，不要忘记用 *chmod* 命令修改文件的执行权限）：

```
./linux-compile.sh
```

**注意** 更多的信息可以在压缩包的 *Readme.txt* 文件中找到。

# 目 录

译者序

前 言

第 1 章 一个简单的 Web 服务器..... 1

1.1 HTTP..... 1

1.1.1 HTTP 请求..... 1

1.1.2 HTTP 响应..... 2

1.2 Socket 类..... 3

1.3 应用程序..... 5

1.3.1 HttpServer 类..... 5

1.3.2 Request 类..... 8

1.3.3 Response 类..... 10

1.3.4 运行应用程序..... 12

1.4 小结..... 13

第 2 章 一个简单的 servlet 容器..... 14

2.1 javax.servlet.Servlet 接口..... 14

2.2 应用程序 1..... 16

2.2.1 HttpServer1 类..... 17

2.2.2 Request 类..... 19

2.2.3 Response 类..... 21

2.2.4 StaticResourceProcessor 类..... 23

2.2.5 ServletProcessor1 类..... 24

2.2.6 运行应用程序..... 27

2.3 应用程序 2..... 27

2.4 小结..... 30

第 3 章 连接器..... 31

3.1 StringManager 类..... 31

3.2 应用程序..... 33

3.2.1 启动应用程序..... 35

3.2.2 HttpURLConnection 类..... 36

3.2.3 创建 HttpRequest 对象..... 38

3.2.4 创建 HttpResponse 对象..... 49

3.2.5 静态资源处理器和 servlet 处理器..... 50

3.2.6 运行应用程序..... 50

3.3 小结..... 52

第 4 章 Tomcat 的默认连接器..... 53

4.1 HTTP 1.1 的新特性..... 54

4.1.1 持久连接..... 54

4.1.2 块编码..... 54

4.1.3 状态码 100 的使用..... 55

4.2 Connector 接口..... 55

4.3 HttpURLConnection 类..... 56

4.3.1 创建服务器套接字..... 56

4.3.2 维护 HttpProcessor 实例..... 56

4.3.3 提供 HTTP 请求服务..... 57

4.4 HttpProcessor 类..... 58

4.5 Request 对象..... 61

4.6 Response 对象..... 62

4.7 处理请求..... 62

4.7.1 解析连接..... 65

4.7.2 解析请求..... 65

4.7.3 解析请求头..... 65

4.8 简单的 Container 应用程序..... 66

4.9 小结..... 70

第 5 章 servlet 容器..... 71

5.1 Container 接口..... 71

5.2 管道任务 .....	73	6.4 LifecycleSupport 类 .....	95
5.2.1 Pipeline 接口 .....	76	6.5 应用程序 .....	97
5.2.2 Valve 接口 .....	76	6.5.1 ex06.pyrmont.core. SimpleContext 类 .....	97
5.2.3 ValveContext 接口 .....	76	6.5.2 ex06.pyrmont.core. SimpleContextLifecycleListener 类 .....	100
5.2.4 Contained 接口 .....	77	6.5.3 ex06.pyrmont.core. SimpleLoader 类 .....	101
5.3 Wrapper 接口 .....	77	6.5.4 ex06.pyrmont.core. SimplePipeline 类 .....	101
5.4 Context 接口 .....	78	6.5.5 ex06.pyrmont.core. SimpleWrapper 类 .....	101
5.5 Wrapper 应用程序 .....	78	6.5.6 运行应用程序 .....	103
5.5.1 ex05.pyrmont.core.SimpleLoader 类 .....	78	6.6 小结 .....	104
5.5.2 ex05.pyrmont.core.SimplePipeline 类 .....	79	第 7 章 日志记录器 .....	105
5.5.3 ex05.pyrmont.core.SimpleWrapper 类 .....	79	7.1 Logger 接口 .....	105
5.5.4 ex05.pyrmont.core. SimpleWrapperValve 类 .....	80	7.2 Tomcat 的日志记录器 .....	106
5.5.5 ex05.pyrmont.valves. ClientIPLoggerValve 类 .....	81	7.2.1 LoggerBase 类 .....	106
5.5.6 ex05.pyrmont.valves. HeaderLoggerValve 类 .....	81	7.2.2 SystemOutLogger 类 .....	107
5.5.7 ex05.pyrmont.startup.Bootstrap1 .....	82	7.2.3 SystemErrLogger 类 .....	107
5.5.8 运行应用程序 .....	84	7.2.4 FileLogger 类 .....	108
5.6 Context 应用程序 .....	84	7.3 应用程序 .....	111
5.6.1 ex05.pyrmont.core. SimpleContextValve 类 .....	87	7.4 小结 .....	112
5.6.2 ex05.pyrmont.core. SimpleContextMapper 类 .....	87	第 8 章 载入器 .....	113
5.6.3 ex05.pyrmont.core. SimpleContext 类 .....	89	8.1 Java 的类载入器 .....	113
5.6.4 ex05.pyrmont.startup.Bootstrap2 .....	89	8.2 Loader 接口 .....	114
5.6.5 运行应用程序 .....	91	8.3 Reloader 接口 .....	116
5.7 小结 .....	92	8.4 WebappLoader 类 .....	116
第 6 章 生命周期 .....	93	8.4.1 创建类载入器 .....	117
6.1 Lifecycle 接口 .....	93	8.4.2 设置仓库 .....	118
6.2 LifecycleEvent 类 .....	94	8.4.3 设置类路径 .....	118
6.3 LifecycleListener 接口 .....	94	8.4.4 设置访问权限 .....	118
		8.4.5 开启新线程执行类的重新载入 .....	118



8.5 WebappClassLoader 类	120	10.6 应用程序	147
8.5.1 类缓存	120	10.6.1 ex10.pyrmont.core. SimpleContextConfig 类	147
8.5.2 载入类	121	10.6.2 ex10.pyrmont.realm. SimpleRealm 类	149
8.5.3 应用程序	121	10.6.3 ex10.pyrmont.realm. SimpleUserDatabaseRealm	152
8.6 运行应用程序	124	10.6.4 ex10.pyrmont.startup. Bootstrap1 类	154
8.7 小结	124	10.6.5 ex10.pyrmont.startup. Bootstrap2 类	156
第 9 章 Session 管理	125	10.6.6 运行应用程序	158
9.1 Session 对象	126	10.7 小结	158
9.1.1 Session 接口	126	第 11 章 StandardWrapper	159
9.1.2 StandardSession 类	127	11.1 方法调用序列	159
9.1.3 StandardSessionFacade 类	129	11.2 SingleThreadModel	160
9.2 Manager	130	11.3 StandardWrapper	161
9.2.1 Manager 接口	130	11.3.1 分配 servlet 实例	162
9.2.2 ManagerBase 类	131	11.3.2 载入 servlet 类	164
9.2.3 StandardManager 类	132	11.3.3 ServletConfig 对象	167
9.2.4 PersistentManagerBase 类	133	11.3.4 servlet 容器的父子关系	169
9.2.5 PersistentManager 类	135	11.4 StandardWrapperFacade 类	170
9.2.6 DistributedManager 类	135	11.5 StandardWrapperValve 类	171
9.3 存储器	136	11.6 FilterDef 类	172
9.3.1 StoreBase 类	137	11.7 ApplicationFilterConfig 类	174
9.3.2 FileStore 类	138	11.8 ApplicationFilterChain 类	175
9.3.3 JDBCStore 类	139	11.9 应用程序	175
9.4 应用程序	139	11.10 小结	177
9.4.1 Bootstrap 类	139	第 12 章 StandardContext 类	178
9.4.2 SimpleWrapperValve 类	140	12.1 StandardContext 的配置	178
9.4.3 运行应用程序	141	12.1.1 StandardContext 类的构造函数	179
9.5 小结	142	12.1.2 启动 StandardContext 实例	179
第 10 章 安全性	143	12.1.3 invoke() 方法	183
10.1 领域	143		
10.2 GenericPrincipal 类	144		
10.3 LoginConfig 类	145		
10.4 Authenticator 接口	145		
10.5 安装验证器阀	146		

12.2 StandardContextMapper 类 .....	184	第 15 章 Digester 库 .....	220
12.3 对重载的支持 .....	187	15.1 Digester 库 .....	221
12.4 backgroundProcess() 方法 .....	188	15.1.1 Digester 类 .....	221
12.5 小结 .....	190	15.1.2 Digester 库示例 1 .....	225
第 13 章 Host 和 Engine .....	191	15.1.3 Digester 库示例 2 .....	227
13.1 Host 接口 .....	191	15.1.4 Rule 类 .....	230
13.2 StandardHost 类 .....	193	15.1.5 Digester 库示例 3 : 使用 RuleSet .....	232
13.3 StandardHostMapper 类 .....	195	15.2 ContextConfig 类 .....	234
13.4 StandardHostValve 类 .....	196	15.2.1 defaultConfig() 方法 .....	236
13.5 为什么必须要有一个 Host 容器 .....	197	15.2.2 applicationConfig() 方法 .....	238
13.6 应用程序 1 .....	198	15.2.3 创建 Web Digester .....	239
13.7 Engine 接口 .....	199	15.3 应用程序 .....	243
13.8 StandardEngine 类 .....	201	15.4 小结 .....	244
13.9 StandardEngineValve 类 .....	201	第 16 章 关闭钩子 .....	245
13.10 应用程序 2 .....	202	16.1 关闭钩子的例子 .....	246
13.11 小结 .....	203	16.2 Tomcat 中的关闭钩子 .....	250
第 14 章 服务器组件和服务组件 .....	204	16.3 小结 .....	250
14.1 服务器组件 .....	204	第 17 章 启动 Tomcat .....	251
14.2 StandardServer 类 .....	206	17.1 Catalina 类 .....	251
14.2.1 initialize() 方法 .....	206	17.1.1 start() 方法 .....	253
14.2.2 start() 方法 .....	207	17.1.2 stop() 方法 .....	256
14.2.3 stop() 方法 .....	207	17.1.3 启动 Digester 对象 .....	256
14.2.4 await() 方法 .....	208	17.1.4 关闭 Digester 对象 .....	258
14.3 Service 接口 .....	209	17.2 Bootstrap 类 .....	259
14.4 StandardService 类 .....	211	17.3 在 Windows 平台上运行 Tomcat .....	264
14.4.1 connector 和 container .....	211	17.3.1 如何编写批处理文件 .....	264
14.4.2 与生命周期有关的方法 .....	213	17.3.2 catalina.bat 批处理文件 .....	267
14.5 应用程序 .....	215	17.3.3 在 Windows 平台上启动 Tomcat .....	276
14.5.1 Bootstrap 类 .....	215	17.3.4 在 Windows 平台上关闭 Tomcat .....	277
14.5.2 Stopper 类 .....	217	17.4 在 Linux 平台上运行 Tomcat .....	278
14.5.3 运行应用程序 .....	218	17.4.1 如何编写 UNIX/Linux Shell 脚本 .....	278
14.6 小结 .....	219	17.4.2 catalina.sh 脚本 .....	283

17.4.3 在 UNIX/Linux 平台上 启动 Tomcat .....	288	第 20 章 基于 JMX 的管理 .....	315
17.4.4 在 UNIX/Linux 平台上 关闭 Tomcat .....	289	20.1 JMX 简介 .....	315
17.5 小结 .....	290	20.2 JMX API .....	316
第 18 章 部署器 .....	291	20.2.1 MBeanServer 类 .....	316
18.1 部署一个 Web 应用程序 .....	291	20.2.2 ObjectName 类 .....	317
18.1.1 部署一个描述符 .....	294	20.3 标准 MBean .....	318
18.1.2 部署一个 WAR 文件 .....	295	20.4 模型 MBean .....	321
18.1.3 部署一个目录 .....	297	20.4.1 MBeanInfo 接口与 ModelMBeanInfo 接口 .....	322
18.1.4 动态部署 .....	297	20.4.2 ModelMBean 示例 .....	323
18.2 Deploy 接口 .....	299	20.5 Commons Modeler 库 .....	326
18.3 StandardHostDeployer 类 .....	302	20.5.1 MBean 描述符 .....	327
18.3.1 安装一个描述符 .....	303	20.5.2 mbean 元素示例 .....	328
18.3.2 安装一个 WAR 文件或目录 .....	304	20.5.3 自己编写一个模型 MBean 类 .....	329
18.3.3 启动 Context 实例 .....	305	20.5.4 Registry 类 .....	329
18.3.4 停止一个 Context 实例 .....	306	20.5.5 ManagedBean .....	329
18.4 小结 .....	306	20.5.6 BaseModelMBean .....	329
第 19 章 Manager 应用程序的 servlet 类 .....	307	20.5.7 使用 Modeler 库 API .....	330
19.1 使用 Manager 应用程序 .....	307	20.6 Catalina 中的 MBean .....	332
19.2 Containerservlet 接口 .....	309	20.6.1 ClassNameMBean 类 .....	333
19.3 初始化 ManagerServlet .....	309	20.6.2 StandardServerMBean 类 .....	333
19.4 列出已经部署的 Web 应用程序 .....	311	20.6.3 MBeanFactory 类 .....	334
19.5 启动 Web 应用程序 .....	312	20.6.4 MBeanUtil .....	335
19.6 关闭 Web 应用程序 .....	313	20.7 创建 Catalina 的 MBean .....	335
19.7 小结 .....	314	20.8 应用程序 .....	339
		20.9 小结 .....	342



# 第 1 章

## 一个简单的 Web 服务器

本章介绍 Java Web 服务器是如何运行的。Web 服务器也称为超文本传输协议 (HyperText Transfer Protocol, HTTP) 服务器, 因为它使用 HTTP 与其客户端 (通常是 Web 浏览器) 进行通信。基于 Java 的 Web 服务器会使用两个重要的类: `java.net.Socket` 类和 `java.net.ServerSocket` 类, 并通过发送 HTTP 消息进行通信。本章先介绍 HTTP 协议和这两个类, 然后介绍一个简单的 Web 服务器。

### 1.1 HTTP

HTTP 允许 Web 服务器和浏览器通过 Internet 发送并接收数据, 是一种基于“请求-响应”的协议。客户端请求一个文件, 服务器端对该请求进行响应。HTTP 使用可靠的 TCP 连接, TCP 协议默认使用 TCP 80 端口。HTTP 协议的第一个版本是 HTTP/0.9, 后来被 HTTP/1.0 取代, 随后 HTTP/1.0 又被当前版本 HTTP/1.1 取代。HTTP/1.1 定义于 RFC (Request for Comment, 请求注解) 2616 中, 可以从 <http://www.w3.org/Protocols/HTTP/1.1/rfc2616.pdf> 下载。

**注意** 本节简单地介绍 HTTP 1.1, 目的是帮助你了解 Web 服务器应用程序发送的消息。若你对此有兴趣, 想了解更多信息, 请阅读 RFC 2616。

在 HTTP 中, 总是由客户端通过建立连接并发送 HTTP 请求来初始化一个事务的。Web 服务器端并不负责联系客户端或建立一个到客户端的回调连接。客户端或服务器端可提前关闭连接。例如, 当使用 Web 浏览器浏览网页时, 可以单击浏览器上的 Stop 按钮来停止下载文件, 这样就有效地关闭了一个 Web 服务器的 HTTP 连接。

#### 1.1.1 HTTP 请求

一个 HTTP 请求包含以下三部分:

- 请求方法——统一资源标识符 (Uniform Resource Identifier, URI) ——协议 / 版本
- 请求头
- 实体

HTTP 请求的示例如下所示:

```
POST /examples/default.jsp HTTP/1.1
Accept: text/plain; text/html
Accept-Language: en-gb
Connection: Keep-Alive
Host: localhost
User-Agent: Mozilla/4.0 (compatible; MSIE 4.01; Windows 98)
```

```
Content-Length: 33
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
```

```
lastName=Franks&firstName=Michael
```

请求方法——URI——协议 / 版本部分出现在请求的第一行，

```
POST /examples/default.jsp HTTP/1.1
```

其中 POST 是请求方法，/examples/default.jsp 表示 URI，HTTP/1.1 表明请求使用的协议及其版本。

每个 HTTP 请求可以使用 HTTP 标准中指定的诸多请求方法中的一种。HTTP 1.1 支持的 7 种请求方法包括：GET、POST、HEAD、OPTIONS、PUT、DELETE 和 TRACE。其中 GET 和 POST 是 Internet 应用中最常用的两种请求方法。

URI 指定 Internet 资源的完整路径。URI 通常会被解释为相对于服务器根目录的相对路径。因此，它总是以 “/” 开头的。统一资源定位符（Uniform Resource Locator，URL）实际上是 URI 的一种类型（参见 <http://www.ietf.org/rfc/rfc2396.txt>）。协议版本指明了当前请求使用的 HTTP 协议的版本。

请求头包含客户端环境和请求实体正文的相关信息。例如，请求头可能会包含浏览器使用的语言，请求实体正文的长度等信息。各个请求头之间用回车 / 换行（Carriage Return/LineFeed，CRLF）符间隔开。

在请求头和请求实体正文之间有一个空行，该空行只有 CRLF 符。这个空行对 HTTP 请求格式非常重要。CRLF 告诉 HTTP 服务器请求实体正文从哪里开始。在有些 Internet 编程书籍中，CRLF 被认为是 HTTP 请求的第 4 部分。

在前面的 HTTP 请求的示例中，请求实体正文很简单，如下所示：

```
lastName=Franks&firstName=Michael
```

当然，在一个典型的 HTTP 请求中，HTTP 请求实体正文也可以很长。

### 1.1.2 HTTP 响应

与 HTTP 请求类似，HTTP 响应也包括三部分：

- 协议——状态码——描述
- 响应头
- 响应实体段

下面是一个 HTTP 响应的示例：

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/4.0
Date: Mon, 5 Jan 2004 13:13:33 GMT
Content-Type: text/html
Last-Modified: Mon, 5 Jan 2004 13:13:12 GMT
Content-Length: 112
```



```
<html>
<head>
<title>HTTP Response Example</title>
</head>
<body>
Welcome to Brainy Software
</body>
</html>
```

HTTP 响应头的第一行与 HTTP 请求头的第一行类似。第一行指明了使用的协议是 HTTP/1.1，请求发送成功（状态码 200 表示请求成功），一切都正常执行。

从上面的示例中可以看出，它与请求头类似，响应头也包含了一些有用的信息。响应实体正文是一段 HTML 代码。响应头和响应实体正文之间由只包含 CRLF 的一个空行分隔。

## 1.2 Socket 类

套接字是网络连接的端点。套接字使应用程序可以从网络中读取数据，可以向网络中写入数据。不同计算机上的两个应用程序可以通过连接发送或接收字节流，以此达到相互通信的目的。为了从一个应用程序向另一个应用程序发送消息，需要知道另一个应用程序中套接字的 IP 地址和端口号。在 Java 中，套接字由 `java.net.Socket` 表示。

要创建一个套接字，可以使用 `Socket` 类中众多构造函数中的一个。其中一个构造函数接收两个参数：主机名和端口号。

```
public Socket (java.lang.String host, int port)
```

其中参数 `host` 是远程主机的名称或 IP 地址，参数 `port` 是连接远程应用程序的端口号。例如，想要通过 80 端口连接 `yahoo.com`，可以使用下面的语句创建 `Socket` 对象：

```
new Socket ("yahoo.com", 80);
```

一旦成功地创建了 `Socket` 类的实例，就可以使用该实例发送或接收字节流。要发送字节流，需要调用 `Socket` 类的 `getOutputStream()` 方法获取一个 `java.io.OutputStream` 对象。要发送文本到远程应用程序，通常需要使用返回的 `OutputStream` 对象创建一个 `java.io.PrintWriter` 对象。若想要从连接的另一端接收字节流，需要调用 `Socket` 类的 `getInputStream()` 方法，该法会返回一个 `java.io.InputStream` 对象。

下面的代码段创建了一个套接字，用于与本地 HTTP 服务器进行通信（127.0.0.1 表示一个本地主机），发送 HTTP 请求，接收服务器的响应信息。以下代码创建了一个 `StringBuffer` 对象来保存响应信息，并将其输出到控制台上。

```
Socket socket = new Socket("127.0.0.1", "8080");
OutputStream os = socket.getOutputStream();
boolean autoflush = true;
PrintWriter out = new PrintWriter(
    socket.getOutputStream(), autoflush);
BufferedReader in = new BufferedReader(
    new InputStreamReader( socket.getInputStream() ));
```

```
// send an HTTP request to the web server
out.println("GET /index.jsp HTTP/1.1");
out.println("Host: localhost:8080");
out.println("Connection: Close");
out.println();
```

```
// read the response
boolean loop = true;
StringBuffer sb = new StringBuffer(8096);
while (loop) {
```

```
    if ( in.ready() ) {
        int i=0;
        while (i!=-1) {
            i = in.read();
            sb.append((char) i);
        }
        loop = false;
    }
```

```
    Thread.currentThread().sleep(50);
}
```

```
// display the response to the out console
System.out.println(sb.toString());
socket.close();
```

**注意** 为了从 Web 服务器上获取正确的响应信息，需要发送一个遵循 HTTP 协议的 HTTP 请求。如果你已经阅读了前一节中关于 HTTP 的描述，你应该已经可以理解以上代码中关于发送 HTTP 请求的方法。

**注意** 可以使用本书中的 `com.brainysoftware.pyrmont.util.HttpSniffer` 类来发送 HTTP 请求，并显示响应信息。要使用该 Java 程序，需要连接到 Internet。但是，要注意的是，防火墙可能会使程序失败。

## ServerSocket 类

`Socket` 类表示一个客户端套接字，即，当想要连接到远程服务器应用程序时创建的套接字。但如果你想要实现一个服务器应用程序（例如一个 HTTP 服务器或 FTP 服务器），你需要另一种方法。因为服务器必须时刻待命，它并不知道客户端应用程序会在什么时候发起连接。正因如此，需要使用 `java.net.ServerSocket` 类，这是服务器套接字的实现。

`ServerSocket` 类与 `Socket` 类并不相同。服务器套接字要等待来自客户端的连接请求。当服务器套接字收到了连接请求后，它会创建一个 `Socket` 实例来处理与客户端的通信。

要创建一个服务器套接字，可以使用 `ServerSocket` 类提供的 4 个构造函数中的任意一个。需要指明 IP 地址和服务器套接字侦听的端口号。典型情况下，IP 地址可以是 127.0.0.1，即服务器套接字会侦听本地机器接收到的连接请求。服务器套接字侦听的 IP 地址称为绑定地址。服务器套接字的另一个重要属性是 `backlog`，后者表示在服务器拒绝接收传入的请求之前，传入的连接请求的最大队列长度。

`ServerSocket` 类的其中一个构造函数的签名如下：

```
public ServerSocket(int port, int backlog, InetAddress bindingAddress);
```

值得注意的是，在这个构造函数中，参数绑定地址必须是 `java.net.InetAddress` 类的实例。创建 `InetAddress` 对象的一种简单方法是调用其静态方法 `getByName()`，传入包含主机名的字符串，代码如下所示：

```
InetAddress.getByName("127.0.0.1");
```

下面的一行代码创建了一个 `ServerSocket` 对象，`ServerSocket` 对象侦听本地主机的 8080 端口，其 backlog 值为 1：

```
new ServerSocket(8080, 1, InetAddress.getByName("127.0.0.1"));
```

创建了 `ServerSocket` 实例后，可以使其等待传入的连接请求，该连接请求会通过服务器套接字侦听的端口上绑定地址传入。这些工作可以通过调用 `ServerSocket` 类的 `accept` 方法完成。只有当接收到连接请求后，该方法才会返回，其返回值是一个 `Socket` 实例。然后，就正如 1.2 节所述，可以使用该 `Socket` 对象与客户端应用程序进行字节流的发送 / 接收。实际上，在本章的应用程序中，`accept` 是唯一会使用到的方法。

## 1.3 应用程序

本章的 Web 服务器应用程序位于 `ex01.pyrmont` 包下，包括三个类：

- `HttpServer`
- `Request`
- `Response`

应用程序的入口点（静态 `main()` 方法）在 `HttpServer` 类中。`main()` 方法创建一个 `HttpServer` 实例，然后，调用其 `await()` 方法。顾名思义，`await()` 方法会在指定端口上等待 HTTP 请求，对其进行处理，然后发送响应信息回客户端。在接收到关闭命令前，它会保持等待状态。

该应用程序仅发送位于指定目录的静态资源的请求，如 HTML 文件和图像文件。它也可以将传入到的 HTTP 请求字节流显示到控制台上。但是，它并不发送任何头信息到浏览器，如日期或 cookies 等。

下面几节将分别展示三个类的具体实现。

### 1.3.1 `HttpServer` 类

`HttpServer` 类表示一个 Web 服务器，具体实现如代码清单 1-1 所示。注意，为了节省空间，`await` 方法在代码清单 1-2 中列出，并没有在代码清单 1-1 中重复。

代码清单 1-1 `HttpServer` 类

```
package ex01.pyrmont;

import java.net.Socket;
import java.net.ServerSocket;
import java.net.InetAddress;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;
import java.io.File;
```



```

public class HttpServer {
    /** WEB_ROOT is the directory where our HTML and other files reside.
     * For this package, WEB_ROOT is the "webroot" directory under the
     * working directory.
     * The working directory is the location in the file system
     * from where the java command was invoked.
     */
    public static final String WEB_ROOT =
        System.getProperty("user.dir") + File.separator + "webroot";

    // shutdown command
    private static final String SHUTDOWN_COMMAND = "/SHUTDOWN";

    // the shutdown command received
    private boolean shutdown = false;

    public static void main(String[] args) {
        HttpServer server = new HttpServer();
        server.await();
    }

    public void await() {
        ...
    }
}

```

这个 Web 服务器可以处理对指定目录中的静态资源的请求，该目录包括由公有静态变量 `final WEB_ROOT` 指明的目录及其所有子目录。`WEB_ROOT` 的初始值为：

```

public static final String WEB_ROOT =
    System.getProperty("user.dir") + File.separator + "webroot";

```

该代码清单包含一个名为 `webroot` 的目录，用于测试该应用程序的一些静态资源都位于该目录下。在该目录下还可以找到用于测试后续章节中应用程序的几个 `servlet` 程序。

若要请求静态资源，可以在浏览器的地址栏或 URL 框中输入如下的 URL：

```
http://machineName:port/staticResource
```

若从另一台机器（不是运行应用程序的那台机器）上向该应用程序发出请求，则 `machineName` 是应用程序所在计算机的名称或 IP 地址；若在同一台机器上发出的请求，则可以将 `machineName` 替换为 `localhost`，此外，连接请求使用的端口为 8080。`staticResource` 是请求的文件的名字，该文件必须位于 `WEB_ROOT` 指向的目录下。

例如，如果你正使用同一台机器来测试该应用程序，你想让 `HttpServer` 对象发送 `index.html` 文件，就可以使用如下的 URL：

```
http://localhost:8080/index.html
```

若要关闭服务器，可以通过 Web 浏览器的地址栏或 URL 框，在 URL 的 `host:port` 部分后面输入预先定义好的字符串，从 Web 浏览器发送一条关闭命令，这样服务器就会收到关闭命令了。关闭命令定义在 `HttpServer` 类的 `SHUTDOWN` 静态 `final` 变量中：

```
private static final String SHUTDOWN_COMMAND = "/SHUTDOWN";
```

因此，若要关闭服务器，需要使用如下的 URL：

`http://localhost:8080/SHUTDOWN`

接下来，看一下代码清单 1-2 中的 `await()` 方法。

代码清单 1-2 `HttpServer` 类的 `await` 方法

```
public void await() {
    ServerSocket serverSocket = null;
    int port = 8080;
    try {
        serverSocket = new ServerSocket(port, 1,
            InetAddress.getByAddress("127.0.0.1"));
    }
    catch (IOException e) {
        e.printStackTrace();
        System.exit(1);
    }
    // Loop waiting for a request
    while (!shutdown) {
        Socket socket = null;
        InputStream input = null;
        OutputStream output = null;

        try {
            socket = serverSocket.accept();
            input = socket.getInputStream();
            output = socket.getOutputStream();
            // create Request object and parse
            Request request = new Request(input);
            request.parse();

            // create Response object
            Response response = new Response(output);
            response.setRequest(request);
            response.sendStaticResource();

            // Close the socket
            socket.close();

            //check if the previous URI is a shutdown command
            shutdown = request.getUri().equals(SHUTDOWN_COMMAND);
        }
        catch (Exception e) {
            e.printStackTrace();
            continue;
        }
    }
}
```

该方法名之所以称为 `await()`，而不是 `wait()`，是因为 `wait()` 方法是 `java.lang.Object` 类中与使用线程相关的重要方法。

`await()` 方法会先创建一个 `ServerSocket` 实例，然后进入一个 `while` 循环：

```

serverSocket = new ServerSocket(port, 1,
    InetAddress.getByName("127.0.0.1"));
...
// Loop waiting for a request
while (!shutdown) {
    ...
}

```

当从 8080 端口接收到 HTTP 请求后，ServerSocket 类的 accept() 方法返回，等待结束：

```
socket = serverSocket.accept();
```

接收到请求后，await() 方法会从 accept() 方法返回的 Socket 实例中获取 java.io.InputStream 对象和 java.io.OutputStream 对象：

```

input = socket.getInputStream();
output = socket.getOutputStream();

```

然后，await() 方法会创建一个 ex01.pyrmont.Request 对象，并调用其 parse() 方法来解析 HTTP 请求的原始数据：

```

// create Request object and parse
Request request = new Request(input);
request.parse();

```

然后，await() 方法会创建一个 Response 对象，并分别调用其 setRequest() 方法和 sendStaticResource() 方法：

```

// create Response object
Response response = new Response(output);
response.setRequest(request);
response.sendStaticResource();

```

最后，await() 方法关闭套接字，调用 Request 类的 getUri() 方法来测试 HTTP 请求的 URI 是否是关闭命令。若是，则将变量 shutdown 设置为 true，程序退出 while 循环。

```

// Close the socket
socket.close();

//check if the previous URI is a shutdown command
shutdown = request.getUri().equals(SHUTDOWN_COMMAND);

```

### 1.3.2 Request 类

ex01.pyrmont.Request 类表示一个 HTTP 请求。可以传递 InputStream 对象（从通过处理与客户端通信的 Socket 对象中获取的），来创建 Request 对象。可以调用 InputStream 对象中的 read() 方法来读取 HTTP 请求的原始数据。

Request 类在代码清单 1-3 中给出。Request 类有两个公共方法（parse() 和 getUri()）和一个私有方法 parseUri()，parse() 方法和 parseUri() 方法分别在代码清单 1-4 和代码清单 1-5 中给出。



代码清单 1-3 Request 类

```

package ex01.pyrmont;

import java.io.InputStream;
import java.io.IOException;

public class Request {
    private InputStream input;
    private String uri;

    public Request(InputStream input) {
        this.input = input;
    }

    public void parse() {
        ...
    }

    private String parseUri(String requestString) {
        ...
    }

    public String getUri() {
        return uri;
    }
}

```

parse() 方法用于解析 HTTP 请求中的原始数据。parse() 方法会调用私有方法 parseUri() 来解析 HTTP 请求的 URI，除此之外，并没有做太多的工作。parseUri() 方法将 URI 存储在变量 uri 中。调用公共方法 getUri() 会返回 HTTP 请求的 URI。

**注意** 处理 HTTP 请求原始数据的方法会在第 3 章和随后章节的应用程序中给出。

为了理解 parse() 和 parseUri() 方法是如何工作的，需要了解 HTTP 请求的结构，这在 1.1 节已经介绍过。在本章中，我们仅仅对 HTTP 请求的第一部分——请求行——感兴趣。请求行以请求方法开始，接着是请求的 URI 和请求所使用的协议及其版本，并以 CRLF 符结束。请求行中的元素以空格分开。例如，使用 GET 方法请求 index.html 文件的请求行如下所示：

```
GET /index.html HTTP/1.1
```

parse() 方法从传入到 Request 对象中的套接字的 InputStream 对象中读取整个字节流，并将字节数组存储在缓冲区中。然后，它使用缓冲区字节数组中的数组填充 StringBuffer 对象 request，并将 StringBuffer 的 String 表示传递给 parseUri() 方法。

parse() 方法已经在代码清单 1-4 中给出。

代码清单 1-4 Request 类的 parse() 方法

```

public void parse() {
    // Read a set of characters from the socket
    StringBuffer request = new StringBuffer(2048);
    int i;
    byte[] buffer = new byte[2048];
    try {

```

```

        i = input.read(buffer);
    }
    catch (IOException e) {
        e.printStackTrace();
        i = -1;
    }
    for (int j=0; j<i; j++) {
        request.append((char) buffer[j]);
    }
    System.out.print(request.toString());
    uri = parseUri(request.toString());
}

```

parseUri() 方法从请求行中获取 URI。代码清单 1-5 展示了 parseUri() 方法的具体实现。parseUri() 方法在请求中搜索第一个和第二个空格，从中找出 URI。

代码清单 1-5 Request 类的 parseUri() 方法

```

private String parseUri(String requestString) {
    int index1, index2;
    index1 = requestString.indexOf(' ');
    if (index1 != -1) {
        index2 = requestString.indexOf(' ', index1 + 1);
        if (index2 > index1)
            return requestString.substring(index1 + 1, index2);
    }
    return null;
}

```

### 1.3.3 Response 类

ex01.pyrmont.Response 类表示 HTTP 响应，其定义如代码清单 1-6 所示。

代码清单 1-6 Response 类的定义

```

package ex01.pyrmont;

import java.io.OutputStream;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.File;

/*
 * HTTP Response = Status-Line
 * (( general-header | response-header | entity-header ) CRLF)
 * CRLF
 * [ message-body ]
 * Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
 */

public class Response {

    private static final int BUFFER_SIZE = 1024;
    Request request;
    OutputStream output;
}

```



```

public Response(OutputStream output) {
    this.output = output;
}

public void setRequest(Request request) {
    this.request = request;
}

public void sendStaticResource() throws IOException {
    byte[] bytes = new byte[BUFFER_SIZE];
    FileInputStream fis = null;
    try {
        File file = new File(HttpServer.WEB_ROOT, request.getUri());
        if (file.exists()) {
            fis = new FileInputStream(file);
            int ch = fis.read(bytes, 0, BUFFER_SIZE);
            while (ch != -1) {
                output.write(bytes, 0, ch);
                ch = fis.read(bytes, 0, BUFFER_SIZE);
            }
        }
        else {
            // file not found
            String errorMessage = "HTTP/1.1 404 File Not Found\r\n" +
                "Content-Type: text/html\r\n" +
                "Content-Length: 23\r\n" +
                "\r\n" +
                "<h1>File Not Found</h1>";
            output.write(errorMessage.getBytes());
        }
    }
    catch (Exception e) {
        // thrown if cannot instantiate a File object
        System.out.println(e.toString());
    }
    finally {
        if (fis != null)
            fis.close();
    }
}
}

```

首先要注意的是 Response 类的构造函数会接收一个 java.io.OutputStream 对象，如下所示：

```

public Response(OutputStream output) {
    this.output = output;
}

```

Response 对象在 HttpServer 类的 await() 方法中通过传入从套接字中获取的 OutputStream 来创建。

Response 类有两个公共方法：setRequest() 和 sendStaticResource()。setRequest() 方法会接收一个 Request 对象为参数。

sendStaticResource() 方法用于发送一个静态资源到浏览器，如 HTML 文件。它首先会通过传入父路径和子路径到 File 类的构造函数中来实例化 java.io.File 类：

```
File file = new File(HttpServer.WEB_ROOT, request.getUri());
```

然后，它检查该文件是否存在。若存在，sendStaticResource() 方法会使用 File 对象创建一个 java.io.FileInputStream 对象。然后它调用 FileInputStream 类的 read() 方法，并将字节数组写入到 OutputStream 输出中。注意，这种情况下，静态资源的内容是作为原始数据发送到浏览器的：

```
if (file.exists()) {
    fis = new FileInputStream(file);
    int ch = fis.read(bytes, 0, BUFFER_SIZE);
    while (ch!=-1) {
        output.write(bytes, 0, ch);
        ch = fis.read(bytes, 0, BUFFER_SIZE);
    }
}
```

若文件不存在，sendStaticResource() 会发送错误消息到浏览器：

```
String errorMessage = "HTTP/1.1 404 File Not Found\r\n" +
    "Content-Type: text/html\r\n" +
    "Content-Length: 23\r\n" +
    "\r\n" +
    "<h1>File Not Found</h1>";
output.write(errorMessage.getBytes());
```

### 1.3.4 运行应用程序

若要运行应用程序，需要在工作目录中执行下面的命令：

```
java ex01.pyrmont.HttpServer
```

若要测试应用程序，可以打开浏览器，在地址栏或 URL 框中输入如下 URL：

```
http://localhost:8080/index.html
```

然后，就可以在浏览器中看到如下的 index.html 页面，如图 1-1 所示：

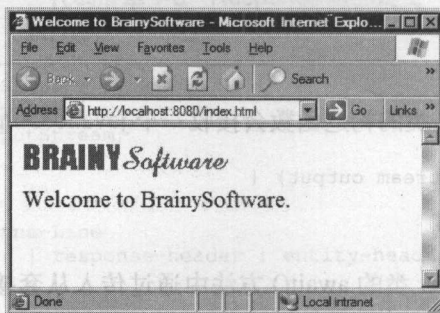


图 1-1 Web 服务器的输出

在控制台中，可以看到类似于如下的 HTTP 请求信息：

```
GET /index.html HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/vnd.ms-excel, application/msword, application/vnd.ms-
powerpoint, application/x-shockwave-flash, application/pdf, */*
Accept-Language: en-us
```

```

Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR
1.1.4322)
Host: localhost:8080
Connection: Keep-Alive

```

```

GET /images/logo.gif HTTP/1.1
Accept: */*
Referer: http://localhost:8080/index.html
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR
1.1.4322)
Host: localhost:8080
Connection: Keep-Alive

```

## 1.4 小结

在本章中，你已经学习到一个简单的 Web 服务器是如何工作的。本章中的应用程序所包含的三个类并不复杂，功能也不完整。但无论如何，这已经是一个不错的学习工具了。第 2 章将会对动态内容的处理过程进行讨论。

2.1 JavaServletServlet 接口

Servlet 编程需要使用到 `javax.servlet.http` 两个包下的接口和类。在 `javax.servlet` 包中，`javax.servlet.Servlet` 接口是最为重要的。所有的 `Servlet` 都必须实现这个接口。这个接口定义了 `Servlet` 类的方法。在 `Servlet` 接口中定义了 3 个方法，如下：

```

public void init(ServletConfig config) throws ServletException
public void service(ServletRequest request, ServletResponse response)
throws ServletException, java.io.IOException
public void destroy()

```

在 `Servlet` 接口中，`init()` 和 `destroy()` 方法是在 `Servlet` 的生命周期中调用的。当 `Servlet` 类被加载到 `Servlet` 容器中时，`init()` 方法就会被调用。在 `Servlet` 容器中，`service()` 方法会被调用。在 `Servlet` 容器中，`destroy()` 方法会被调用。在 `Servlet` 容器中，`init()` 方法会被调用。在 `Servlet` 容器中，`service()` 方法会被调用。在 `Servlet` 容器中，`destroy()` 方法会被调用。



## 第 ② 章

# 一个简单的 servlet 容器

本章通过两个小应用程序说明如何开发自己的 servlet 容器。第一个应用程序的设计非常简单，仅仅用于说明 servlet 容器是如何运行的。它然后演变为第二个 servlet 容器，后者稍微复杂一点。

**注意** 每一章的应用程序示例中的 servlet 容器都是在前一章的基础上进行演化而成，在第 17 章中，会构建出一个功能齐全的 Tomcat servlet 容器。

这两个 servlet 容器都能处理简单的 servlet 和静态资源。PrimitiveServlet 类可用于测试 servlet 容器。PrimitiveServlet 类的定义在代码清单 2-1 中给出，其类文件位于 webroot 目录下。更复杂一点的 servlet 会超出本章中的 servlet 容器的处理能力，但在后面章节中会介绍如何构建更复杂一点的 servlet 容器。

两个应用程序所使用的类都在 ex02.pyrmont 包下。为了更好地理解应用程序的运行机制，你需要熟悉 javax.servlet.Servlet 接口。下一节会对 javax.servlet.Servlet 接口进行详细说明。然后，你会学习到 servlet 容器是如何为一个 servlet 的 HTTP 请求提供服务的。

### 2.1 javax.servlet.Servlet 接口

Servlet 编程需要使用到 javax.servlet 和 javax.servlet.http 两个包下的接口和类。在所有的类和接口中，javax.servlet.servlet 接口是最为重要。所有的 servlet 程序都必须实现该接口或继承自实现了该接口的类。

在 Servlet 接口中声明了 5 个方法，方法签名如下：

```
public void init(ServletConfig config) throws ServletException
public void service(ServletRequest request, ServletResponse response)
    throws ServletException, java.io.IOException
public void destroy()
public ServletConfig getServletConfig()
public java.lang.String getServletInfo()
```

在 Servlet 接口中声明的 5 个方法里，init()、service() 和 destroy() 方法是与 servlet 的生命周期相关的方法。当实例化某个 servlet 类后，servlet 容器会调用其 init() 方法进行初始化。servlet 容器只会调用该方法一次，调用后则可以执行服务方法了。在 servlet 接收任何请求之前，必须是经过正确初始化的。servlet 程序员可以覆盖此方法，在其中编写仅需要执行一次的初始化代码，例如载入数据库驱动程序、初始化默认值等。一般情况下，init() 方法可以留空。

当 servlet 的一个客户端请求到达后，servlet 容器就调用相应的 servlet 的 service() 方

法，并将 `javax.servlet.ServletRequest` 对象和 `javax.servlet.ServletResponse` 对象作为参数传入。`ServletRequest` 对象包含客户端的 HTTP 请求的信息，`ServletResponse` 对象则封装 servlet 的响应信息。在 servlet 对象的整个生命周期内，`service()` 方法会被多次调用。

在将 servlet 实例从服务中移除前，servlet 容器会调用 servlet 实例的 `destroy()` 方法。一般当 servlet 容器关闭或 servlet 容器要释放内存时，才会将 servlet 实例移除，而且只有当 servlet 实例的 `service()` 方法中的所有线程都退出或执行超时后，才会调用 `destroy()` 方法。当 servlet 容器调用了某个 servlet 实例的 `destroy()` 方法后，它就不会再调用该 servlet 实例的 `service()` 方法了。调用 `destroy()` 方法让 servlet 对象有机会去清理自身持有的资源，如内存、文件句柄和线程等，确保所有的持久化状态与内存中该 servlet 对象的当前状态同步。

代码清单 2-1 展示了名为 `PrimitiveServlet` 的 servlet 的代码，该 servlet 非常简单，可以用来测试本章中的 servlet 容器应用程序。`PrimitiveServlet` 类实现了 `javax.servlet.Servlet` 接口（所有的 servlet 都要实现该接口），提供了 `Servlet` 接口中声明的 5 个方法的实现。`PrimitiveServlet` 类所做的事情非常简单。每次调用 `init()`、`service()` 和 `destroy()` 方法时，`Servlet` 都会将方法名写入标准控制台中。此外，`service()` 方法会从 `ServletResponse` 对象中获取 `java.io.PrintWriter` 对象，并将字符串发送到客户端浏览器。

代码清单 2-1 `PrimitiveServlet.java` 类的定义

```
import javax.servlet.*;
import java.io.IOException;
import java.io.PrintWriter;

public class PrimitiveServlet implements Servlet {

    public void init(ServletConfig config) throws ServletException {
        System.out.println("init");
    }

    public void service(ServletRequest request, ServletResponse response)
        throws ServletException, IOException {
        System.out.println("from service");
        PrintWriter out = response.getWriter();
        out.println("Hello. Roses are red.");
        out.print("Violets are blue.");
    }

    public void destroy() {
        System.out.println("destroy");
    }

    public String getServletInfo() {
        return null;
    }

    public ServletConfig getServletConfig() {
        return null;
    }
}
```

## 2.2 应用程序 1

下面从 servlet 容器的角度审视 servlet 程序的开发。简单来说, 对一个 Servlet 的每个 HTTP 请求, 一个功能齐全的 servlet 容器有以下几件事要做:

- 当第一次调用某个 servlet 时, 要载入该 servlet 类, 并调用其 `init()` 方法 (仅此一次);
- 针对每个 request 请求, 创建一个 `javax.servlet.ServletRequest` 实例和一个 `javax.servlet.ServletResponse` 实例;
- 调用该 servlet 的 `service()` 方法, 将 `ServletRequest` 对象和 `ServletResponse` 对象作为参数传入;
- 当关闭该 servlet 类时, 调用其 `destroy()` 方法, 并卸载该 servlet 类。

本章所要建立的 servlet 容器是一个很小的容器, 没有实现所有的功能。因此, 它只能运行非常简单的 servlet, 而且也会不调用 servlet 的 `init()` 和 `destroy()` 方法。它会做以下几件事:

- 等待 HTTP 请求;
- 创建一个 `ServletRequest` 对象和一个 `ServletResponse` 对象;
- 若请求的是一个静态资源, 则调用 `StaticResourceProcessor` 对象的 `process()` 方法, 传入 `ServletRequest` 对象和 `ServletResponse` 对象;
- 若请求的是 servlet, 则载入相应的 servlet 类, 调用其 `service()` 方法, 传入 `ServletRequest` 对象和 `ServletResponse` 对象。

**注意** 在该 servlet 容器中, 每次请求 servlet 都会载入相应的 servlet 类。

本节的应用程序包括 6 个类:

- `HttpServer1`

- `Request`

- `Response`

- `StaticResourceProcessor`

- `ServletProcessor1`

- `Constants`

图 2-1 展示了本节中的 servlet 容器的 UML 类图。

该应用程序的入口点 (静态 `main()` 方法) 在类 `HttpServer1` 中。`main()` 方法创建 `HttpServer1` 的一个实例, 然后调用其 `await()` 方法。`await()` 方法会等待 HTTP 请求, 为接收到的每个请求创建一个 `Request` 和一个 `Response` 对象, 并根据该 HTTP 请求的是静态资源或是 servlet, 将该 HTTP 请求分发给一个 `StaticResourceProcessor` 实例或一个 `ServletProcessor` 实例。

`Constants` 类中定义了静态 `final WEB_ROOT`, 供其他的类引用。`WEB_ROOT` 指定了该 servlet 容器中使用的 `PrimitiveServlet` 类和静态资源的位置。

`HttpServer1` 类的实例会一直等待 HTTP 请求, 直到接收到一条关闭命令。可以使用第 1 章介绍的方法来发布关闭命令。

该应用程序中的各个类会在接下来的几节中逐个说明。



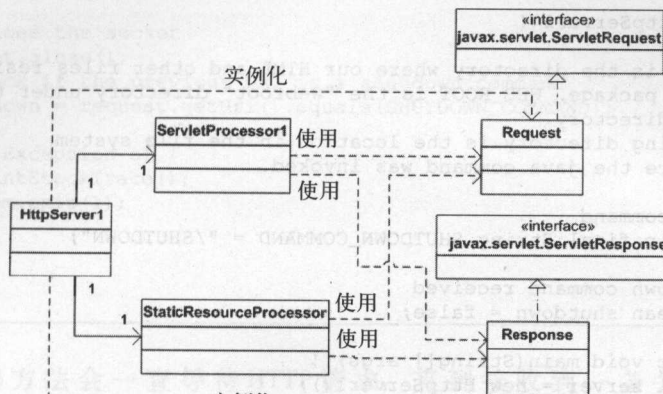


图 2-1 第一个 servlet 容器的 UML 类图

## 2.2.1 HttpServer1 类

应用程序 1 中的 HttpServer1 类与第 1 章中简单 Web 服务器应用程序中的 HttpServer 类似。但是，该应用程序中的 HttpServer1 类既可以对静态资源请求，也可以对于 servlet 资源请求。若要请求一个静态资源，可以在浏览器的地址栏或 URL 框中输入如下格式的 URL：

```
http://machineName:port/staticResource
```

这与第 1 章的 Web 服务器应用程序中对静态资源的请求相同。

若要请求 servlet 资源，可以使用如下格式的 URL：

```
http://machineName:port/servlet/servletClass
```

因此，若要请求本地浏览器上的名为 PrimitiveServlet 的 servlet，可以在浏览器的地址栏或 URL 框中输入如下的 URL：

```
http://localhost:8080/servlet/Primitiveservlet
```

应用程序 1 中的 servlet 容器会处理对 PrimitiveServlet 的请求。但是，若要调用其他的 servlet（如 ModernServlet），则 servlet 容器抛出异常。在后面的章节中，你将学会如何构建可以兼具两种功能的 servlet 容器。

HttpServer1 类的定义在代码清单 2-2 中。

代码清单 2-2 HttpServer1 类的 await() 方法

```
package ex02.pyrmont;

import java.net.Socket;
import java.net.ServerSocket;
import java.net.InetAddress;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;
```

```

public class HttpServer1 {

    /** WEB_ROOT is the directory where our HTML and other files reside.
     * For this package, WEB_ROOT is the "webroot" directory under the
     * working directory.
     * The working directory is the location in the file system
     * from where the java command was invoked.
     */
    // shutdown command
    private static final String SHUTDOWN_COMMAND = "/SHUTDOWN";

    // the shutdown command received
    private boolean shutdown = false;

    public static void main(String[] args) {
        HttpServer1 server = new HttpServer1();
        server.await();
    }

    public void await() {
        ServerSocket serverSocket = null;
        int port = 8080;
        try {
            serverSocket = new ServerSocket(port, 1,
                InetAddress.getBy_name("127.0.0.1"));
        }
        catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
        }

        // Loop waiting for a request
        while (!shutdown) {
            Socket socket = null;
            InputStream input = null;
            OutputStream output = null;
            try {
                socket = serverSocket.accept();
                input = socket.getInputStream();
                output = socket.getOutputStream();

                // create Request object and parse
                Request request = new Request(input);
                request.parse();

                // create Response object
                Response response = new Response(output);
                response.setRequest(request);

                // check if this is a request for a servlet or
                // a static resource
                // a request for a servlet begins with "/servlet/"
                if (request.getUri().startsWith("/servlet/")) {
                    ServletProcessor1 processor = new ServletProcessor1();
                    processor.process(request, response);
                }
                else {
                    StaticResourceProcessor processor =
                        new StaticResourceProcessor();
                    processor.process(request, response);
                }
            }
        }
    }
}

```



```

        // Close the socket
        socket.close();
        //check if the previous URI is a shutdown command
        shutdown = request.getUri().equals(SHUTDOWN_COMMAND);
    }
    catch (Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
}
}
}

```

该类的 `await()` 方法会一直等待 HTTP 请求，直到接收到一条关闭命令，这点与第 1 章中的 `await()` 方法类似。区别在于，本章中的 `await()` 方法可以将 HTTP 请求分发给 `StaticResourceProcessor` 对象或 `ServletProcessor` 对象来处理。当 URI 包含字符串 `“/servlet/”` 时，会把请求转发给 `ServletProcessor` 对象处理。否则的话，把 HTTP 请求传递给 `StaticResourceProcessor` 对象处理。注意代码清单 2-2 中灰色的部分。

## 2.2.2 Request 类

servlet 的 `service` 方法会从 servlet 容器中接收一个 `javax.servlet.ServletRequest` 实例一个和 `javax.servlet.ServletResponse` 实例。即，对每个 HTTP 请求来说，servlet 容器必须创建一个 `ServletRequest` 对象和一个 `ServletResponse` 对象，并将它们作为参数传给它服务的 servlet 的 `service()` 方法。

`ex02.pyrmont.Request` 类表示被传递给 servlet 的 `service()` 方法的一个 request 对象。它必须实现 `javax.Servlet.ServletRequest` 接口中声明的所有方法。但为了简单起见，这里只给出了部分方法的实现，其余方法的实现会在后面的章节给出。为了能够编译 `Request` 类，需要将未实现的方法留空。代码清单 2-3 中给出了 `Request` 类的定义，其签名返回 `object` 实例的所有方法都会返回 `null`。

代码清单 2-3 Request 类

```

package ex02.pyrmont;

import java.io.InputStream;
import java.io.IOException;
import java.io.BufferedReader;
import java.io.UnsupportedEncodingException;
import java.util.Enumeration;
import java.util.Locale;
import java.util.Map;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletInputStream;
import javax.servlet.ServletRequest;

public class Request implements ServletRequest {

    private InputStream input;
    private String uri;

```

```

public Request(InputStream input) {
    this.input = input;
}

public String getUri() {
    return uri;
}

private String parseUri(String requestString) {
    int index1, index2;
    index1 = requestString.indexOf(' ');
    if (index1 != -1) {
        index2 = requestString.indexOf(' ', index1 + 1);
        if (index2 > index1)
            return requestString.substring(index1 + 1, index2);
    }
    return null;
}

public void parse() {
    // Read a set of characters from the socket
    StringBuffer request = new StringBuffer(2048);
    int i;
    byte[] buffer = new byte[2048];
    try {
        i = input.read(buffer);
    }
    catch (IOException e) {
        e.printStackTrace();
        i = -1;
    }
    for (int j=0; j<i; j++) {
        request.append((char) buffer[j]);
    }
    System.out.print(request.toString());
    uri = parseUri(request.toString());
}

/* implementation of ServletRequest */
public Object getAttribute(String attribute) {
    return null;
}

public Enumeration getAttributeNames() {
    return null;
}

public String getRealPath(String path) {
    return null;
}

public RequestDispatcher getRequestDispatcher(String path) {
    return null;
}

public boolean isSecure() {
    return false;
}

public String getCharacterEncoding() {
    return null;
}

public int getContentLength() {
    return 0;
}

```

```

public String getContentType() {
    return null;
}
public ServletInputStream getInputStream() throws IOException {
    return null;
}
public Locale getLocale() {
    return null;
}
public Enumeration getLocales() {
    return null;
}
public String getParameter(String name) {
    return null;
}
public Map getParameterMap() {
    return null;
}
public Enumeration getParameterNames() {
    return null;
}
public String[] getParameterValues(String parameter) {
    return null;
}
public String getProtocol() {
    return null;
}
public BufferedReader getReader() throws IOException {
    return null;
}
public String getRemoteAddr() {
    return null;
}
public String getRemoteHost() {
    return null;
}
public String getScheme() {
    return null;
}
public String getServerName() {
    return null;
}
public int getServerPort() {
    return 0;
}
public void removeAttribute(String attribute) {
}
public void setAttribute(String key, Object value) {
}
public void setCharacterEncoding(String encoding)
    throws UnsupportedEncodingException {
}
}

```

此外, Request 类还包括了在第 1 章中介绍过的 parse() 和 getUri() 方法。

## 2.2.3 Response 类

ex02.pyrmont.Response 类实现 javax.servlet.ServletResponse 接口, 类定义参见代码清单 2-4。该类提供了 ServletResponse 接口中声明的所有方法的实现。与 Request 类类似, 除了 getWriter() 方法以外, 大部分方法的实现都留空。



## 代码清单 2-4 Response 类

```

package ex02.pyrmont;

import java.io.OutputStream;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.File;
import java.io.PrintWriter;
import java.util.Locale;
import javax.servlet.ServletResponse;
import javax.servlet.ServletOutputStream;

public class Response implements ServletResponse {

    private static final int BUFFER_SIZE = 1024;
    Request request;
    OutputStream output;
    PrintWriter writer;

    public Response(OutputStream output) {
        this.output = output;
    }

    public void setRequest(Request request) {
        this.request = request;
    }

    /* This method is used to serve static pages */
    public void sendStaticResource() throws IOException {
        byte[] bytes = new byte[BUFFER_SIZE];
        FileInputStream fis = null;
        try {
            /* request.getUri has been replaced by request.getRequestURI */
            File file = new File(Constants.WEB_ROOT, request.getUri());
            fis = new FileInputStream(file);
            /*
            HTTP Response = Status-Line
            *(( general-header | response-header | entity-header ) CRLF)
            CRLF
            [ message-body ]
            Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
            */
            int ch = fis.read(bytes, 0, BUFFER_SIZE);
            while (ch != -1) {
                output.write(bytes, 0, ch);
                ch = fis.read(bytes, 0, BUFFER_SIZE);
            }
        } catch (FileNotFoundException e) {
            String errorMessage = "HTTP/1.1 404 File Not Found\r\n" +
                "Content-Type: text/html\r\n" +
                "Content-Length: 23\r\n" +
                "\r\n" +
                "<h1>File Not Found</h1>";
            output.write(errorMessage.getBytes());
        } finally {
            if (fis != null)
                fis.close();
        }
    }
}

```

```

/** implementation of ServletResponse */
public void flushBuffer() throws IOException { }
public int getBufferSize() {
    return 0;
}
public String getCharacterEncoding() {
    return null;
}
public Locale getLocale() {
    return null;
}
public ServletOutputStream getOutputStream() throws IOException {
    return null;
}
public PrintWriter getWriter() throws IOException {
    // autoflush is true, println() will flush,
    // but print() will not.
    writer = new PrintWriter(output, true);
    return writer;
}
public boolean isCommitted() {
    return false;
}
public void reset() { }
public void resetBuffer() { }
public void setBufferSize(int size) { }
public void setContentLength(int length) { }
public void setContentType(String type) { }
public void setLocale(Locale locale) { }
}

```

在 `getWriter()` 方法中, `PrintWriter` 类的构造函数的第 2 个参数是一个布尔值, 表示是否启用 `autoFlush`。对第 2 个参数传入 `true` 表示对 `println()` 方法的任何调用都会刷新输出。但是调用 `print()` 方法不会刷新输出。

因此, 如果在 `servlet` 的 `service()` 方法的最后一行调用 `print()` 方法, 则该输出内容不会被发送给浏览器。这个 bug 会在后续的版本中修改。

`Response` 类中仍然保留了第 1 章中介绍过的 `sendStaticResource()` 方法。

## 2.2.4 StaticResourceProcessor 类

`ex02.pyrmont.StaticResourceProcessor` 类用于处理对静态资源的请求。该类只有一个方法, 即 `process()` 方法。代码清单 2-5 给出了 `StaticResourceProcessor` 类的定义。

代码清单 2-5 `StaticResourceProcessor` 类的定义

```

package ex02.pyrmont;

import java.io.IOException;

public class StaticResourceProcessor {

    public void process(Request request, Response response) {
        try {
            response.sendStaticResource();
        }
    }
}

```

```

        catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

process() 方法接收两个参数：一个 `ex02.pyrmont.Request` 实例和一个 `ex02.pyrmont.Response` 实例。该方法仅仅调用 `Response` 对象的 `sendStaticResource()` 方法。

### 2.2.5 ServletProcessor1 类

`ex02.pyrmont.ServletProcessor1` 类的定义参见代码清单 2-6，该类用于处理对 `Servlet` 资源的 HTTP 请求。

代码清单 2-6 ServletProcessor1 类的定义

```

package ex02.pyrmont;

import java.net.URL;
import java.net.URLClassLoader;
import java.net.URLStreamHandler;
import java.io.File;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class ServletProcessor1 {

    public void process(Request request, Response response) {
        String uri = request.getUri();
        String servletName = uri.substring(uri.lastIndexOf("/") + 1);
        URLClassLoader loader = null;
        try {
            // create a URLClassLoader
            URL[] urls = new URL[1];
            URLStreamHandler streamHandler = null;
            File classPath = new File(Constants.WEB_ROOT);
            // the forming of repository is taken from the
            // createClassLoader method in
            // org.apache.catalina.startup.ClassLoaderFactory
            String repository =
                (new URL("file", null, classPath.getCanonicalPath() +
                    File.separator).toString());
            // the code for forming the URL is taken from
            // the addRepository method in
            // org.apache.catalina.loader.StandardClassLoader.
            urls[0] = new URL(null, repository, streamHandler);
            loader = new URLClassLoader(urls);
        }
        catch (IOException e) {
            System.out.println(e.toString());
        }
    }
}

```



```
Class myClass = null;
try {
    myClass = loader.loadClass(servletName);
}
catch (ClassNotFoundException e) {
    System.out.println(e.toString());
}
```

```
Servlet servlet = null;
```

```
try {
    servlet = (Servlet) myClass.newInstance();
    servlet.service((ServletRequest) request,
        (ServletResponse) response);
}
catch (Exception e) {
    System.out.println(e.toString());
}
catch (Throwable e) {
    System.out.println(e.toString());
}
```

ServletProcessor1 类很简单，只有一个方法：process() 方法。该方法接收两个参数，一个 javax.servlet.ServletRequest 实例和一个 javax.servlet.ServletResponse 实例。该方法通过调用 getRequestUri() 方法从 ServletRequest 对象中获取 URI：

```
String uri = request.getUri();
```

记住，URI 的格式如下所示：

```
/servlet/servletName
```

其中，servletName 是请求的 servlet 资源的类名。

为了载入 servlet 类，需要从 URI 中获取 servlet 的类名。可以使用 process() 方法的下一行语句获取 servlet 的类名：

```
String servletName = uri.substring(uri.lastIndexOf("/") + 1);
```

接下来，process() 方法会载入该 servlet 类。为了载入类，需要创建一个类载入器，并且指明到哪里查找要载入的类。对于本节的 servlet 容器，类载入器会到 Constant.WEB\_ROOT 指定的工作目录下的 webroot 目录中查找要载入的类。

**注意** 有关类载入器的详细内容将在第 8 章中介绍。

为了载入一个 servlet 类，可以使用 java.net.URLClassLoader 类来完成，该类是 java.lang.ClassLoader 类的一个直接子类。一旦创建了 URLClassLoader 类的实例后，就可以使用它的 loadClass() 方法来载入 servlet 类。实例化 URLClassLoader 类很简单。该类有三个构造函数，其中比较简单的一个构造函数的签名如下所示：

```
public URLClassLoader(URL[] urls);
```

其中, `urls` 是一个 `java.net.URL` 对象数组, 当载入一个类时每个 `URL` 对象都指明了类载入器要到哪里查找类。若一个 `URL` 以 “/” 结尾, 则表明它指向的是一个目录。否则, `URL` 默认指向一个 `JAR` 文件, 根据需要载入器会下载并打开这个 `JAR` 文件。

**注意** 在 `servlet` 容器中, 类载入器查找 `servlet` 类的目录称为仓库 (repository)。

在应用程序中, 类载入器只需要查找一个位置, 即工作目录下的 `webroot` 目录。因此, 需要先创建只有一个 `URL` 的一个数组。`URL` 类提供了一系列构造函数, 因此有很多种方法可以创建 `URL` 对象。对于本应用程序, 使用与 `Tomcat` 中另一个类中使用的相同构造函数, 该构造函数的签名如下所示:

```
public URL(URL context, java.lang.String spec, URLStreamHandler handler)
    throws MalformedURLException
```

可以为第 2 个参数指定一个目录, 指定第 1 个和第 3 个参数为 `null`, 这样就可以使用构造函数了。但是还有一个构造函数, 它接受 3 个参数:

```
public URL(java.lang.String protocol, java.lang.String host, java.lang.String
file) throws MalformedURLException
```

因此, 若只使用如下语句, 编译器就无法知道要调用哪个构造函数了, 并且会报错:

```
new URL(null, aString, null);
```

因此, 可以使用下面的代码, 对于编译器指明第三个参数的类型:

```
URLStreamHandler streamHandler = null;
new URL(null, aString, streamHandler);
```

第 2 个参数中的字符串指明了仓库的路径, 也就是查找 `servlet` 类的目录。可以使用下面的代码生成仓库:

```
String repository = (new URL("file", null, classPath.getCanonicalPath() + File.
separator)).toString();
```

将上述代码综合到一起, 就得到了创建 `URLClassLoader` 实例的 `process()` 方法的部分代码:

```
// create a URLClassLoader
URL[] urls = new URL[1];
URLStreamHandler streamHandler = null;
File classPath = new File(Constants.WEB_ROOT);
String repository = (new URL("file", null,
    classPath.getCanonicalPath() + File.separator)).toString();
urls[0] = new URL(null, repository, streamHandler);
loader = new URLClassLoader(urls);
```

**注意** 生成仓库后会调用 `org.apache.catalina.startup.ClassLoaderFactory` 类的 `createClassLoader()` 方法, 生成 `URL` 对象后会调用 `org.apache.catalina.loader.StandardClassLoader` 类的

addRepository() 方法。这些方法将在后续章节中介绍。

有了类载入器后，就可以通过调用 loadClass() 方法来载入 servlet 类：

```
Class myClass = null;
try {
    myClass = loader.loadClass(servletName);
}
catch (ClassNotFoundException e) {
    System.out.println(e.toString());
}
```

接下来，process() 方法会创建已载入的 servlet 类的一个实例，将其向下转型为 javax.servlet.Servlet，并调用其 service() 方法：

```
Servlet servlet = null;
try {
    servlet = (Servlet) myClass.newInstance();
    servlet.service((ServletRequest) request,
        (ServletResponse) response);
}
catch (Exception e) {
    System.out.println(e.toString());
}
catch (Throwable e) {
    System.out.println(e.toString());
}
```

## 2.2.6 运行应用程序

要在 Windows 平台上运行该程序，可以在工作目录下执行如下命令：

```
java -classpath ./lib/servlet.jar;./ ex02.pyrmont.HttpServer1
```

在 Linux 平台上，需要用冒号分割两个库文件：

```
java -classpath ./lib/servlet.jar:./ ex02.pyrmont.HttpServer1
```

若想测试应用程序，可以在浏览器的地址栏或者 URL 框中输入如下地址：

```
http://localhost:8080/index.html
```

或

```
http://localhost:8080/servlet/Primitiveservlet
```

当调用 PrimitiveServlet 类时，可以在浏览器中看到如下输出：

```
Hello. Roses are red.
```

注意，你是看不到第 2 个字符串 “Violets are blue” 的，因为只有第 1 个字符串会发送到浏览器。这个问题将在第 3 章解决。

## 2.3 应用程序 2

应用程序 1 中有一个严重的问题。在 ServletProcessor1 类的 process() 方法中，必须将 ex02.



pyrmont.Request 的实例向上转型为 javax.servlet.ServletRequest 实例，将 ex02.pyrmont.Response 实例向上转型为 javax.servlet.ServletResponse 实例，然后将它们作为参数传递给 servlet 的 service() 方法：

```
try {
    servlet = (Servlet) myClass.newInstance();
    servlet.service((ServletRequest) request,
        (ServletResponse) response);
}
```

这是不安全的做法，了解这个 servlet 容器内部工作原理的 servlet 程序员可以将 ServletRequest 实例和 ServletResponse 实例分别向下转型为 ex02.pyrmont.Request 实例和 ex02.pyrmont.Response 实例，就可以调用它们各自的公共方法了。有了 Request 实例后，就可以调用其 parse() 方法；而有了 Response 实例后，就可以调用其 sendStaticResource() 方法。

不能将 parse() 方法和 sendStaticResource() 方法设置为私有方法，因为它们会被其他的类调用，但是这两个方法在 servlet 中不应该是可用的，所以这个方法不好。一种解决方法是将 Request 类和 Response 类都设为默认的访问修饰符，这样就不能从 ex02.pyrmont 包外对它们进行访问了。但是，这里有一个更完美的方法：使用外观类。UML 关系图如图 2-2 所示。

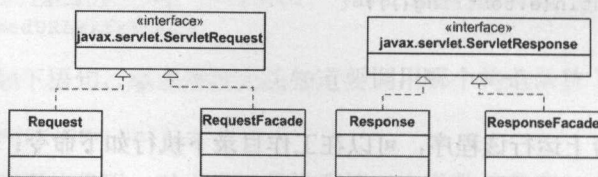


图 2-2 外观类

在第 2 个应用程序中，添加了两个外观类：RequestFacade 和 ResponseFacade。RequestFacade 类实现 ServletRequest 接口，在其构造函数中需要把它指定的一个 Request 对象传递给 ServletRequest 对象引用。ServletRequest 接口中每个方法的实现都会调用 Request 对象的相应方法。但是，ServletRequest 对象本身是私有的，无法从类的外部进行访问。相比于将 Request 对象向上转型为 ServletRequest 对象，并将其传给 service() 方法，这里会创建一个 RequestFacade 对象，再将其传递给 service() 方法。servlet 程序员仍然可以将 servletRequest 实例向下转型为 RequestFacade 对象，但它们只能访问 ServletRequest 接口中提供的方法。现在 parse() 方法和 getUri() 方法是安全的了。

代码清单 2-7 给出了 RequestFacade 类的定义。

代码清单 2-7 RequestFacade 类

```
package ex02.pyrmont;
public class RequestFacade implements ServletRequest {
    private ServletRequest request = null;

    public RequestFacade(Request request) {
        this.request = request;
    }

    /* implementation of the ServletRequest*/
}
```

```

public Object getAttribute(String attribute) {
    return request.getAttribute(attribute);
}

public Enumeration getAttributeNames() {
    return request.getAttributeNames();
}

...

```

注意 RequestFacade 类的构造函数，它接受一个 Request 对象，但立即将其赋给私有的 ServletRequest 对象引用。还要注意的，RequestFacade 中的每个方法会调用 ServletRequest 对象中相对应的方法来执行。

ResponseFacade 类的情况与此相同。

应用程序 2 中共需要使用 6 个类：

- HttpServer2
- Request
- Response
- StaticResourceProcessor
- ServletProcessor2
- Constants

HttpServer2 类与 HttpServer1 类相似，只是在其 await() 方法中它会使用 ServletProcessor2 类，而不是 ServletProcessor1 类：

```

if (request.getUri().startsWith("/servlet/")) {
    ServletProcessor2 processor = new ServletProcessor2();
    processor.process(request, response);
}
else {
    ...
}

```

servletProcessor2 类与 servletProcessor1 类相似，只是在其 process() 方法的以下部分中有些许不同：

```

Servlet servlet = null;
RequestFacade requestFacade = new RequestFacade(request);
ResponseFacade responseFacade = new ResponseFacade(response);
try {
    servlet = (Servlet) myClass.newInstance();
    servlet.service((ServletRequest) requestFacade,
        (ServletResponse) responseFacade);
}

```

## 运行应用程序

要在 Windows 平台上运行该程序，可以在工作目录下执行如下命令：

```
java -classpath ./lib/servlet.jar;./ ex02.pyrmont.HttpServer2
```

在 Linux 平台上, 需要用冒号分割两个库文件:

```
java -classpath ./lib/servlet.jar:./ ex02.pyrmont.HttpServer2
```

可以使用与应用程序 1 相同的 URL 来测试, 结果是相同的。

## 2.4 小结

本章总讨论了两个 servlet 容器, 它们既可以处理静态资源请求, 也可以处理像 `PrimitiveServlet` 一样简单的 servlet 资源请求, 并给出了与 `javax.servlet.Servlet` 接口及其相关类型的一些背景信息。



## 第③章

# 连接器

正如前言所述, Catalina 中有两个主要的模块, 连接器 (connector) 和容器 (container)。在本章中, 将会建立一个连接器来增强第 2 章中的应用程序的功能, 用一种更好的方法来创建 request 对和 response 对象。兼容 Servlet 2.3 和 2.4 规范的连接器必须要负责创建 javax.servlet.http.HttpServletRequest 对象和 javax.servlet.http.HttpServletResponse 的实例, 并将它们作为 servlet 的 service 方法的参数传入。在第 2 章中, servlet 容器仅仅能运行实现了 javax.servlet.Servlet 接口的 servlet 容器, 并把 javax.servlet.ServletRequest 实例和 javax.servlet.ServletResponse 实例传递给 service 方法。因为连接器并不知道 servlet 对象的类型 (即不知道该 servlet 对象是实现了 javax.servlet.Servlet 接口, 还是继承自 javax.servlet.GenericServlet 类或 javax.servlet.http.HttpServlet 类), 连接器总是会提供 HttpServletRequest 实例和 HttpServletResponse 实例。

在本章的应用程序中, 连接器解析 HTTP 请求头, 使 servlet 实例能够获取到请求头、cookie 和请求参数 / 值等信息。可以修改第 2 章中 Response 类的 getWriter() 方法, 使其可以工作得更好些。有了这些增强功能之后, 就可以从 PrimitiveServlet 实例中获得完整的响应信息了, 从而可以运行更加复杂一点的 servlet 类 (例如 ModernServlet)。

本章中所要建立的连接器实际上是 Tomcat 4 中的默认连接器的简化版, 默认连接器将会在第 4 章中讨论。虽然在 Tomcat 4 中已经以不再推荐使用默认连接器了, 但它却是一个很好的学习工具。在本章剩下的部分中, “连接器” 特指要在本章的应用程序中建立的模块。

**注意** 与前面几章中的应用程序不同, 在本章的应用程序中, 连接器和容器是分开的。

本章的应用程序都在 ex03.pyrmont 包及其子包下。其中连接器所使用的类都在 ex03.pyrmont.connector 包及 ex03.pyrmont.connector.http 包下。从本章的应用程序开始, 每章的应用程序中都会有一个启动类, 用来启动整个应用程序。但是, 另一方面, 还没有一种机制来关闭应用程序。一旦应用程序运行后, 就只能通过关闭控制台 (在 Windows 平台上) 或终止进程 (在 UNIX/Linux 平台上) 的方式强制关闭应用程序。

在对本章的应用程序进行说明之前, 会先对 org.apache.catalina.util 包下的 StringManager 类进行说明。该类用来处理应用程序中不同模块和 Catalina 本身中错误消息的国际化操作。在此之后, 再对应用程序进行说明。

### 3.1 StringManager 类

像 Tomcat 这样的大型应用程序必须小心仔细地处理错误消息。在 Tomcat 中, 错误消息对系统管理员和 servlet 程序员来说都是很有用的。例如, 系统管理员可以很容易地根据 Tomcat 的

错误日志消息定位到发生异常的位置。而对于 servlet 程序员来说，在抛出的每个 `javax.servlet.ServletException` 异常中，Tomcat 都会发送一条特殊的错误消息，这样，程序员就可以知道 servlet 程序到底哪里出错了。

Tomcat 处理错误消息的方法是将错误消息存储在一个 `properties` 文件中，便于读取和编辑。但是 Tomcat 中有几百个类。若是将所有类使用的错误消息都存储在一个大的 `properties` 属性文件中，并维护这个文件将会是一场噩梦。为了避免这种情况，Tomcat 将 `properties` 文件划分到不同的包中。例如，`org.apache.catalina.connector` 包下的 `properties` 属性文件包含该包中任何类可能抛出的所有的异常消息。每个 `properties` 文件都是用 `org.apache.catalina.util.StringManager` 类的一个实例来处理的。当 Tomcat 运行时，会产生 `StringManager` 类的很多实例，每个实例都会读取某个包下的指定 `properties` 文件。此外，由于 Tomcat 非常受欢迎，因此对错误消息进行国际化处理也是有必要的，当前共有三种语言得到支持。使用英文版错误消息的 `properties` 文件命名为 `LocalStrings.properties`。其他两种语言是西班牙语和日语，错误消息文件分别名为 `LocalStrings_es.properties` 和 `LocalStrings_ja.properties`。

当包中的某个类需要在其包内的 `properties` 文件中查找错误消息时，它会先获取对应的 `StringManager` 实例。但是，同一个包下的许多类会使用同一个 `StringManager` 实例，若是为每个要查找错误消息的对象创建一个 `StringManger` 实例是很浪费资源的。因此，设计 `StringManager` 类以便 `StringManager` 类的实例被包内所有的对象共享。若你对设计模式比较熟悉的话，你可能已经猜到了，`StringManager` 是单例类。`StringManager` 只有一个私有的构造函数，这样就不能在类的外部通过关键字 `new` 来实例化它了。只能通过调用其公共静态方法 `getManager()` 来获得其实例，该方法需要一个指明了包名的参数。每个 `StringManager` 实例都会以这个包名作为其键，存储在一个 `Hashtable` 中。

```
private static Hashtable managers = new Hashtable();
public synchronized static StringManager
getManager(String packageName) {
    StringManager mgr = (StringManager)managers.get(packageName);
    if (mgr == null) {
        mgr = new StringManager(packageName);
        managers.put(packageName, mgr);
    }
    return mgr;
}
```

**注意** 在附加的 ZIP 文件中有一篇名为“The Singleton Pattern”的文章，其中会对单例模式进行讲解。

例如，要想从 `ex03.pyrmont.connector.http` 包下的类中使用 `StringManager`，需要将包名字符串作为参数传到 `StringManager` 类的 `getManager()` 方法中：

```
StringManager sm =
    StringManager.getManager("ex03.pyrmont.connector.http");
```

在 `ex03.pyrmont.connector.http` 包下，可以找到三个 `properties` 文件，`LocalStrings.properties`、`LocalStrings_es.properties` 和 `LocalStrings_ja.properties`。`StringManager` 实例会根据运行该应用程

序的服务器的语言环境来选择使用哪个文件。若打开 LocalStrings.properties 文件, 第 1 个非注释行的内容如下:

```
httpConnector.alreadyInitialized=HTTP connector has already been initialized
```

要想获取错误消息，可以使用 `StringManager` 类的 `getString()` 方法，该方法需要传入一个错误码。其中一个重载方法的签名如下所示：

```
public String getString(String key) ;
```

调用 `getString()` 方法，并传入上述的“`httpConnector.alreadyInitialized`”错误码时，会得到错误消息“HTTP connector has already been initialized”。

### 3.2 应用程序

从本章开始，每章的应用程序都会按照模块进行划分。本章的应用程序包含3个模块：连接器模块、启动模块和核心模块。

启动模块只有一个类 (Bootstrap)，后者负责启动应用程序。连接器模块中的类可分为以下 5 个类型：

- 连接器及其支持类 (HttpConnector 和 HttpProcessor);
- 表示 HTTP 请求的类 (HttpRequest) 及其支持类;
- 表示 HTTP 响应的类 (HttpResponse) 及其支持类;
- 外观类 (HttpRequestFacade 和 HttpResponseFacade);
- 常量类。

核心模块包含两个类，ServletProcessor 类和 StaticResourceProcessor 类。

图 3-1 展示了本章中应用程序的 UML 类图。为了使类图更容易理解，一些与 `HttpRequest` 类和 `HttpResponse` 类相关的类被省略掉。在讨论 `Request` 对象和 `Response` 对象时，你会看到它们各自的 UML 类图。

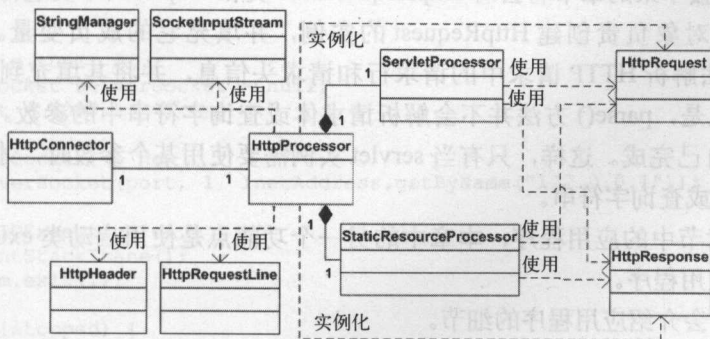


图 3-1 本章应用程序的 UML 类图

该图相比于图 2-1 中的图，第 2 章中的 `HttpServer` 类分成 `HttpConnector` 和 `HttpProcessor` 两个类，`Request` 类和 `Response` 类分别被 `HttpRequest` 和 `HttpResponse` 代替。此外，本章的应用程序中还使用了一些其他的类。



在第2章中, `HttpServer` 类负责等待 HTTP 请求, 并创建 `Request` 和 `Response` 对象。在本章的应用程序中, 等待 HTTP 请求的工作由 `HttpConnector` 实例完成, 创建 `Request` 和 `Response` 对象的工作由 `HttpProcessor` 实例完成。

在本章中, HTTP 请求对象使用 `HttpRequest` 类表示, `HttpRequest` 类实现 `javax.servlet.http.HttpServletRequest` 接口。`HttpRequest` 对象会被转型为 `HttpServletRequest` 对象, 然后作为参数传递给被调用的 `servlet` 实例的 `service()` 方法。因此, 必须正确地设置每个 `HttpRequest` 实例的成员变量供 `servlet` 实例使用。需要设置的值包括: URI、查询字符串、参数、Cookie 和其他一些请求头信息等。因为连接器并不知道被调用的 `servlet` 会使用哪些变量, 所以连接器必须解析从 HTTP 请求中获取的所有信息。但是, 解析 HTTP 请求涉及一些系统开销大的字符串操作以及一些其他操作。若是连接器仅仅解析会被 `servlet` 实例用到的值它就会节省很多 CPU 周期。比如, 如果 `servlet` 实例不会使用任何请求参数 (即它不会调用 `javax.servlet.http.HttpServletRequest` 类的 `getParameter()`、`getParameterMap()`、`getParameterNames()` 或 `getParameterValues()` 方法), 连接器就不需要从查询字符串和 / 或 HTTP 请求体中解析这些参数。在这些参数被 `servlet` 实例真正调用前, Tomcat 的默认连接器 (包括本章应用程序中的连接器) 是不会解析它们的, 这样就可以使程序执行得更有效率。

Tomcat 中的默认连接器和本章应用程序中的连接器使用 `SocketInputStream` 类从套接字的 `InputStream` 对象中读取字节流。`SocketInputStream` 实例是 `java.io.InputStream` 实例的包装类, `SocketInputStream` 实例可以通过调用套接字的 `getInputStream()` 方法获得。`SocketInputStream` 类提供了两个重要的方法, 分别是 `readRequestLine()` 和 `readHeader()`。`readRequestLine()` 方法会返回一个 HTTP 请求中第 1 行的内容, 即包含了 URI、请求方法和 HTTP 版本信息的内容。由于从套接字的输入流中处理字节流是指读取从第 1 个字节到最后 1 个字节 (无法从后向前读取) 的内容, 因此 `readRequestLine()` 方法必须在 `readHeader()` 方法之前调用。每次调用 `readHeader()` 方法都会返回一个名 / 值对, 所以应重复调用 `readHeader()` 方法, 直到读取了所有的请求头信息。`readRequestLine()` 方法的返回值是一个 `HttpRequestLine` 实例, `readHeader()` 方法的返回值是一个 `HttpHeader` 对象。接下来的章节将会对 `HttpRequestLine` 类和 `HttpHeader` 类进行讨论。

`HttpProcessor` 对象负责创建 `HttpRequest` 的实例, 并填充它的成员变量。`HttpProcessor` 类使用其 `parse()` 方法解析 HTTP 请求中的请求行和请求头信息, 并将其填充到 `HttpRequest` 对象的成员变量中。但是, `parse()` 方法并不会解析请求体或查询字符串中的参数。这个任务由各个 `HttpRequest` 对象自己完成。这样, 只有当 `servlet` 实例需要使用某个参数时, 才会由 `HttpRequest` 对象去解析请求体或查询字符串。

相比与前面章节中的应用程序, 本章中的另一个功能点是使用启动类 `ex03.pyrmont.startup.Bootstrap` 来启动应用程序。

下面的几节将会介绍应用程序的细节。

- 启动应用程序
- 连接器
- 创建 `HttpRequest` 对象
- 创建 `HttpResponse` 对象

- 静态资源处理器和 servlet 处理器
- 运行应用程序

### 3.2.1 启动应用程序

本章中的应用程序是通过 `ex03.pyrmont.startup.Bootstrap` 类来启动的。该类的定义在代码清单 3-1 中给出。

代码清单 3-1 Bootstrap 类

```
package ex03.pyrmont.startup;
import ex03.pyrmont.connector.http.HttpConnector;
public final class Bootstrap {
    public static void main(String[] args) {
        HttpConnector connector = new HttpConnector();
        connector.start();
    }
}
```

`Bootstrap` 类的 `main()` 方法通过实例化 `HttpConnector` 类，并调用其 `start()` 方法就可以启动应用程序。`HttpConnector` 类的定义在代码清单 3-2 中给出。

代码清单 3-2 HttpConnector 类

```
package ex03.pyrmont.connector.http;

import java.io.IOException;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;

public class HttpConnector implements Runnable {
    boolean stopped;
    private String scheme = "http";

    public String getScheme() {
        return scheme;
    }

    public void run() {
        ServerSocket serverSocket = null;
        int port = 8080;
        try {
            serverSocket = new
                ServerSocket(port, 1, InetAddress.getByName("127.0.0.1"));
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
        }
        while (!stopped) {
            // Accept the next incoming connection from the server socket
            Socket socket = null;
            try {
                socket = serverSocket.accept();
            }
        }
    }
}
```

```
        catch (Exception e) {  
            continue;  
        }  
        // Hand this socket off to an HttpProcessor  
        HttpProcessor processor = new HttpProcessor(this);  
        processor.process(socket);  
    }  
}  
  
public void start() {  
    Thread thread = new Thread(this);  
    thread.start();  
}  
}
```

### 3.2.2 HttpURLConnection 类

连接器是 `ex03.pyrmont.connector.http.HttpConnector` 类的实例，负责创建一个服务器套接字，该套接字会等待传入的 HTTP 请求。`HttpConnector` 类已在代码清单 3-2 中给出。

`HttpConnector` 类实现了 `java.lang.Runnable` 接口。当你启动应用程序时，会创建一个 `HttpConnector` 实例，该实例另起一个线程来运行。

`HttpConnector` 类实现 `java.lang.Runnable`，这样它可以专用于自己的线程。当启动应用程序时，创建 `HttpConnector` 的一个实例并执行它的 `run()` 方法。

**注意** 你可以阅读文章“Working with Threads”来学习如何创建 Java 线程的相关知识。

`run()` 方法包含一个 `while` 循环，在循环中会执行如下三个操作：

- 等待 HTTP 请求；
- 为每个请求创建一个 `HttpProcessor` 实例；
- 调用 `HttpProcessor` 对象的 `process()` 方法。

**注意** `HttpConnector` 类的 `run()` 方法其实与第 2 章中 `HttpServer1` 类的 `await()` 方法实现了类似的功能。

你可以看到，实际上，`HttpConnector` 类与 `ex02.pyrmont.HttpServer1` 类非常相似，区别在于从 `java.net.ServerSocket` 类的 `accept()` 方法中获取一个套接字，创建一个 `HttpProcessor` 实例并传入该套接字，调用其 `process()` 方法。

**注意** `HttpConnector` 类有一个 `getScheme()` 方法，后者会返回请求协议（如 HTTP 协议）。

`HttpProcessor` 类的 `process()` 方法接收来自传入的 HTTP 请求的套接字。对每个传入的 HTTP 请求，它要完成 4 个操作：

- 创建一个 `HttpRequest` 对象；
- 创建一个 `HttpResponse` 对象；
- 解析 HTTP 请求的第 1 行内容和请求头信息，填充 `HttpRequest` 对象；



- 将 `HttpRequest` 对象和 `HttpResponse` 对象传递给 `ServletProcessor` 或 `StaticResourceProcessor` 的 `process()` 方法。与第 2 章中相同, `ServletProcessor` 类会调用请求的 `Servlet` 实例的 `service()` 方法, `StaticResourceProcessor` 类会将请求的静态资源发送给客户端。

`HttpProcessor` 类的 `process()` 方法的实现在代码清单 3-3 中给出。

代码清单 3-3 `HttpProcessor` 类的 `process()` 方法

```
public void process(Socket socket) {
    SocketInputStream input = null;
    OutputStream output = null;
    try {
        input = new SocketInputStream(socket.getInputStream(), 2048);
        output = socket.getOutputStream();

        // create HttpRequest object and parse
        request = new HttpRequest(input);

        // create HttpResponse object
        response = new HttpResponse(output);
        response.setRequest(request);

        response.setHeader("Server", "Pyrmont Servlet Container");

        parseRequest(input, output);
        parseHeaders(input);

        //check if this is a request for a servlet or a static resource
        //a request for a servlet begins with "/servlet/"
        if (request.getRequestURI().startsWith("/servlet/")) {
            ServletProcessor processor = new ServletProcessor();
            processor.process(request, response);
        }
        else {
            StaticResourceProcessor processor = new
                StaticResourceProcessor();
            processor.process(request, response);
        }

        // Close the socket
        socket.close();
        // no shutdown for this application
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

`process()` 方法首先会获取套接字的输入流和输出流。但请注意, 在这种方法中, 使用了继承自 `java.io.InputStream` 的 `SocketInputStream` 类:

```
SocketInputStream input = null;
OutputStream output = null;
try {
    input = new SocketInputStream(socket.getInputStream(), 2048);
    output = socket.getOutputStream();
```

然后, 它会创建一个 `HttpRequest` 实例和一个 `HttpResponse` 实例, 然后将 `HttpRequest` 实例

赋值给 `HttpResponse` 实例:

```
// create HttpRequest object and parse
request = new HttpRequest(input);

// create HttpResponse object
response = new HttpResponse(output);
response.setRequest(request);
```

在本章的应用程序中, `HttpResponse` 类比第2章中的 `Response` 类稍微复杂一点。首先, 可以调用 `HttpResponse` 类的 `setHeader()` 方法向客户端发送响应头信息:

```
response.setHeader("Server", "Pyrmont Servlet Container");
```

其次, `process()` 方法会调用 `HttpProcessor` 类的两个私有方法来解析请求:

```
parseRequest(input, output);
parseHeaders(input);
```

然后, 它会根据请求的 URI 模式来判断, 将 `HttpRequest` 对象和 `HttpResponse` 对象发送给 `ServletProcessor` 类或 `StaticResourceProcessor` 类来处理:

```
if (request.getRequestURI().startsWith("/servlet/")) {
    ServletProcessor processor = new ServletProcessor();
    processor.process(request, response);
}
else {
    StaticResourceProcessor processor =
        new StaticResourceProcessor();
    processor.process(request, response);
}
```

最后, 它关闭套接字:

```
socket.close();
```

注意, `HttpProcessor` 类使用了 `org.apache.catalina.util.StringManager` 类来发送错误消息:

```
protected StringManager sm =
    StringManager.getManager("ex03.pyrmont.connector.http");
```

调用 `HttpProcessor` 类的私有方法——`parseRequest()`、`parseHeaders()` 和 `normalize()`——来帮助填充 `HttpRequest` 对象。这些方法将在 3.3.3 节中进行讨论。

### 3.2.3 创建 `HttpRequest` 对象

`HttpRequest` 类实现了 `javax.servlet.http.HttpServletRequest` 接口。其外观类是 `HttpRequestFacade` 类。图 3-2 展示了 `HttpRequest` 类及其相关类的 UML 类图。

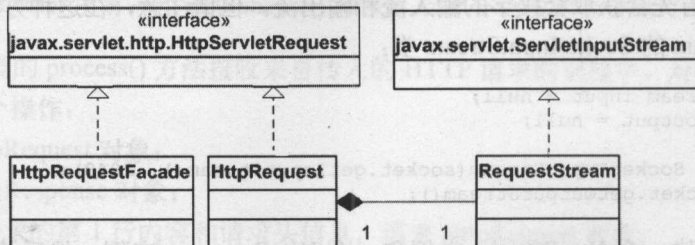


图 3-2 `HttpRequest` 类及其相关类的 UML 类图

其中 `HttpRequest` 类的很多方法都是空方法（在第 4 章中会提供完整实现），但是 `servlet` 程序员已经可以从中获取引入的 HTTP 请求的请求头、Cookie 和请求参数等信息了。这三类数据分别存储在如下引用变量中：

```
protected HashMap headers = new HashMap();
protected ArrayList cookies = new ArrayList();
protected ParameterMap parameters = null;
```

**注意** `ParameterMap` 类将会在本节下面的“5. 获取参数”节中介绍。

因此，`servlet` 程序员就可以通过调用 `javax.servlet.http.HttpServletRequest` 类的一些方法获取 HTTP 请求信息，这些方法包括 `getCookies()`、`getDateHeader()`、`getHeader()`、`getHeaderNames()`、`getHeaders()`、`getParameter()`、`getPrameterMap()`、`getParameterNames()` 和 `getParameterValues()`。当使用正确的值填充请求头、Cookie 和请求参数后，相关方法的实现就比较简单了，就像在 `HttpRequest` 类中看到的那样。

当然，这里最主要的工作是解析 HTTP 请求并填充 `HttpRequest` 对象。对每个请求头和 Cookie，`HttpRequest` 类都提供了 `addHeader()` 方法和 `addCookie()` 方法来填充相关信息，这两个方法会由 `HttpProcessor` 类的 `parseHeaders()` 方法进行调用。当真正需要用到请求参数时，才会使用 `HttpRequest` 类的 `parseParameters()` 方法解析请求参数。相关方法将会在本节中介绍。

解析 HTTP 请求相对来说比较复杂，所以下面将分成 5 个小节来进行说明：

- 读取套接字的输入流
- 解析请求行
- 解析请求头
- 解析 Cookie
- 获取参数

#### 1. 读取套接字的输入流

在第 1 章和第 2 章中，`ex01.pyrmont.HttpRequest` 类和 `ex02.pyrmont.HttpRequest` 类对 HTTP 请求进行过初步的解析。通过调用 `java.io.InputStream` 类的 `read()` 方法，从请求行中获取到了方法、URI、HTTP 协议版本等信息：

```
byte[] buffer = new byte[2048];
try {
    // input is the InputStream from the socket.
    i = input.read(buffer);
}
```

不需要试图进一步解析这两个应用程序的请求。在本章的应用程序中，将会使用 `ex03.pyrmont.connector.http.SocketInputStream` 类来进行解析，该类实际上就是 `org.apache.catalina.connector.http.SocketInputStream` 类的一个副本。该类提供了一些方法来获取请求行和请求头信息。

需要传入一个 `InputStream` 对象和一个指明缓冲区大小的整数来创建一个 `SocketInputStream` 实例。在本章的应用程序中，在 `ex03.pyrmont.connector.http.HttpProcessor` 类的 `process()` 方法中会创建一个 `SocketInputStream` 对象，如以下代码段所示：



```

SocketInputStream input = null;
OutputStream output = null;
try {
    input = new SocketInputStream(socket.getInputStream(), 2048);
    ...
}

```

如前所述，之所以使用 `SocketInputStream` 类就是为了调用其 `readRequestLine()` 方法和 `readHeader()` 方法。

## 2. 解析请求行

`HttpProcessor` 类的 `process()` 方法会调用私有方法 `parseRequest()` 来解析请求行，即 HTTP 请求的第 1 行内容。下面是一个 HTTP 请求行的示例：

```
GET /myApp/Modernservlet?userName=tarzan&password=pwd HTTP/1.1
```

请求行的第 2 部分是 URI 加上一个可选的查询字符串。在这个例子中，URI 是：

```
/myApp/Modernservlet
```

问号后面的部分就是查询字符串，如下所示：

```
userName=tarzan&password=pwd
```

查询字符串可以包含 0 个或多个请求参数。在上面的例子中，包含了两个名/值对，`userName/tarzan` 和 `password/pwd`。在 `servlet/JSP` 编程中，参数名 `jsessionid` 用于携带一个会话标识符。会话标识符通常是作为 Cookie 嵌入的，但是当浏览器禁用了 Cookie 时，程序员也可以将会话标识符嵌入到查询字符串中。

当从 `HttpProcessor` 类的 `process()` 方法调用 `parseRequest()` 方法时，`request` 变量会指向一个 `HttpRequest` 实例。`parseRequest()` 方法解析请求行，从而获取一些值，并将其赋给 `HttpRequest` 对象。代码清单 3-4 展示了 `parseRequest()` 方法的实现。

代码清单 3-4 `HttpProcessor` 类中 `parseRequest()` 方法的实现

```

private void parseRequest(SocketInputStream input, OutputStream output)
    throws IOException, ServletException {

    // Parse the incoming request line
    input.readRequestLine(requestLine);
    String method =
        new String(requestLine.method, 0, requestLine.methodEnd);
    String uri = null;
    String protocol = new String(requestLine.protocol, 0,
        requestLine.protocolEnd);

    // Validate the incoming request line
    if (method.length() < 1) {
        throw new ServletException("Missing HTTP request method");
    }
    else if (requestLine.uriEnd < 1) {
        throw new ServletException("Missing HTTP request URI");
    }

    // Parse any query parameters out of the request URI
    int question = requestLine.indexOf("?");
    if (question >= 0) {
        request.setQueryString(new String(requestLine.uri, question + 1,

```

```

        requestLine.uriEnd - question - 1));
        uri = new String(requestLine.uri, 0, question);
    }
    else {
        request.setQueryString(null);
        uri = new String(requestLine.uri, 0, requestLine.uriEnd);
    }

    // Checking for an absolute URI (with the HTTP protocol)
    if (!uri.startsWith("/")) {
        int pos = uri.indexOf("://");
        // Parsing out protocol and host name
        if (pos != -1) {
            pos = uri.indexOf('/', pos + 3);
            if (pos == -1) {
                uri = "";
            }
            else {
                uri = uri.substring(pos);
            }
        }
    }

    // Parse any requested session ID out of the request URI
    String match = ";jsessionid=";
    int semicolon = uri.indexOf(match);
    if (semicolon >= 0) {
        String rest = uri.substring(semicolon + match.length());
        int semicolon2 = rest.indexOf(';');
        if (semicolon2 >= 0) {
            request.setRequestedSessionId(rest.substring(0, semicolon2));
            rest = rest.substring(semicolon2);
        }
        else {
            request.setRequestedSessionId(rest);
            rest = "";
        }
        request.setRequestedSessionURL(true);
        uri = uri.substring(0, semicolon) + rest;
    }
    else {
        request.setRequestedSessionId(null);
        request.setRequestedSessionURL(false);
    }

    // Normalize URI (using String operations at the moment)
    String normalizedUri = normalize(uri);
    // Set the corresponding request properties
    ((HttpRequest) request).setMethod(method);
    request.setProtocol(protocol);
    if (normalizedUri != null) {
        ((HttpRequest) request).setRequestURI(normalizedUri);
    }
    else {
        ((HttpRequest) request).setRequestURI(uri);
    }
    if (normalizedUri == null) {
        throw new ServletException("Invalid URI: " + uri + "'");
    }
}

```

`parseRequest()` 方法首先会调用 `SocketInputStream` 类的 `readRequestLine()` 方法:

```
input.readRequestLine(requestLine);
```

其中, 变量 `requestLine` 是 `HttpProcessor` 中的一个 `HttpRequestLine` 实例:

```
private HttpRequestLine requestLine = new HttpRequestLine();
```

调用其 `readRequestLine()` 方法, 使用 `SocketInputStream` 对象中的信息填充 `HttpRequestLine` 实例。

接着, `parseRequest()` 方法从请求行中获取请求方法、URI 和请求协议的版本信息:

```
String method =
    new String(requestLine.method, 0, requestLine.methodEnd);
String uri = null;
String protocol = new String(requestLine.protocol, 0,
    requestLine.protocolEnd);
```

但是, 在 URI 后面可能会有一个查询字符串。若有, 则查询字符串与 URI 是用一个问号分隔的。因此, `parseRequest()` 方法会首先调用 `HttpRequest` 类的 `setQueryString()` 方法来获取查询字符串, 并填充 `HttpRequest` 对象:

```
// Parse any query parameters out of the request URI
int question = requestLine.indexOf("?");
if (question >= 0) { // there is a query string.
    request.setQueryString(new String(requestLine.uri, question + 1,
        requestLine.uriEnd - question - 1));
    uri = new String(requestLine.uri, 0, question);
}
else {
    request.setQueryString(null);
    uri = new String(requestLine.uri, 0, requestLine.uriEnd);
}
```

但是, 大多数的 URI 都指向一个相对路径中的资源, 当然 URI 也可以是一个绝对路径中的值, 例如:

```
http://www.brainysoftware.com/index.html?name=Tarzan
```

`parseRequest()` 方法会进行如下检查:

```
// Checking for an absolute URI (with the HTTP protocol)
if (!uri.startsWith("/")) {
    // not starting with /, this is an absolute URI
    int pos = uri.indexOf("://");
    // Parsing out protocol and host name
    if (pos != -1) {
        pos = uri.indexOf('/', pos + 3);
        if (pos == -1) {
            uri = "";
        }
        else {
            uri = uri.substring(pos);
        }
    }
}
```



然后, 查询字符串可能也会包含一个会话标识符, 参数名为 `jsessionid`。因此, `parseRequest()` 方法还要检查是否包含会话标识符。若在查询字符串中包含 `jsessionid`, 则 `parseRequest()` 方法要获取 `jsessionid` 的值, 并调用 `HttpRequest` 类的 `setRequestedSessionId()` 方法填充 `HttpRequest` 实例:

```
// Parse any requested session ID out of the request URI
String match = ";jsessionid=";
int semicolon = uri.indexOf(match);
if (semicolon >= 0) {
    String rest = uri.substring(semicolon + match.length());
    int semicolon2 = rest.indexOf(';');
    if (semicolon2 >= 0) {
        request.setRequestedSessionId(rest.substring(0, semicolon2));
        rest = rest.substring(semicolon2);
    }
    else {
        request.setRequestedSessionId(rest);
        rest = "";
    }
    request.setRequestedSessionURL(true);
    uri = uri.substring(0, semicolon) + rest;
}
else {
    request.setRequestedSessionId(null);
    request.setRequestedSessionURL(false);
}
```

若存在参数 `jsessionid`, 则表明会话标识符在查询字符串中, 而不在 Cookie 中。因此, 需要调用该请求的 `setRequestSessionURL()` 方法并传入 `true` 值。否则, 调用 `setRequestSessionURL()` 方法并传入 `false` 值, 同时调用 `setRequestedSessionURL()` 方法并传入 `null`。

此刻, `jsessionid` 已经不包含 `uri` 的值。

然后, `parseRequest()` 方法会将 `URI` 传入到 `normalize()` 方法中, 对非正常的 `URL` 进行修正。例如, 出现 “\” 的地方会被替换为 “/”。若 `URI` 本身是正常的, 或不正常的地方可以修正, 则 `normalize()` 方法会返回相同的 `URI` 或修正过的 `URI`。若 `URI` 无法修正, 则会认为它是无效的, `normalize()` 方法返回 `null`。在这种情况下 (`normalize()` 方法返回 `null`), `parseRequest()` 方法会在方法的末尾抛出异常。

最后, `parseRequest()` 方法会设置 `HttpRequest` 对象的一些属性:

```
((HttpRequest) request).setMethod(method);
request.setProtocol(protocol);
if (normalizedUri != null) {
    ((HttpRequest) request).setRequestURI(normalizedUri);
}
else {
    ((HttpRequest) request).setRequestURI(uri);
}
```

另外, 若 `normalize()` 方法返回 `null`, 该方法会抛出异常:

```
if (normalizedUri == null) {
    throw new ServletException("Invalid URI: " + uri + "");
}
```

### 3. 解析请求头

请求头信息由一个 `HttpHeader` 类表示。该类将在第 4 章中详细讨论, 但在这里, 有以下 5 件事需要了解:

- 可以通过其类的无参构造函数来创建一个 `HttpHeader` 实例;
- 创建了 `HttpHeader` 实例后, 可以将其传给 `SocketInputStream` 类的 `readHeader()` 方法。若有请求头信息可以读取, `readHeader()` 方法会相应地填充 `HttpHeader` 对象。若没有请求头信息可以读取, 则 `HttpHeader` 实例的 `nameEnd` 和 `valueEnd` 字段都会是 0;
- 要获取请求头的名字和值, 可以使用如下的方法:

```
String name = new String(header.name, 0, header.nameEnd);
String value = new String(header.value, 0, header.valueEnd);
```

`parseHeaders()` 方法中有一个 `while` 循环, 后者不断地从 `SocketInputStream` 中读取请求头信息, 直到全部读完。在循环开始时, 会先创建一个 `HttpHeader` 实例, 然后将其传给 `SocketInputStream` 类的 `readHeader()` 方法:

```
HttpHeader header = new HttpHeader();
// Read the next header
input.readHeader(header);
```

然后, 可以通过检查 `HttpHeader` 实例的 `nameEnd` 和 `valueEnd` 字段来判断是否已经从输入流中读取了所有的请求头信息:

```
if (header.nameEnd == 0) {
    if (header.valueEnd == 0) {
        return;
    }
    else {
        throw new ServletException
            (sm.getString("httpProcessor.parseHeaders.colon"));
    }
}
```

若还有请求头没有读取, 可以使用下面的方法获取请求头的名称和值:

```
String name = new String(header.name, 0, header.nameEnd);
String value = new String(header.value, 0, header.valueEnd);
```

当获取了请求头的名称和值之后, 就可以调用 `HttpRequest` 对象的 `addHeader()` 方法, 将其添加到 `HttpRequest` 对象的 `HashMap` 请求头中:

```
request.addHeader(name, value);
```

某些请求头包含一些属性设置信息, 例如, 当调用 `javax.servlet.ServletRequest` 类的 `getContentTypeLength()` 方法时, 会返回请求头 “content-length” 的值, 而请求头 “cookies” 中是一些 `Cookie` 的集合。下面是相关的处理过程:

```
if (name.equals("cookie")) {
    ... // process cookies here
}
else if (name.equals("content-length")) {
    int n = -1;
    try {
        n = Integer.parseInt(value);
    }
    catch (Exception e) {
        throw new ServletException(sm.getString(
            "httpProcessor.parseHeaders.contentLength"));
    }
    request.setContentLength(n);
```

```

}
else if (name.equals("content-type")) {
    request.setContentType(value);
}
}

```

对 Cookie 的解析将在下一小节中详细讨论。

#### 4. 解析 Cookie

Cookie 是由浏览器作为 HTTP 请求头的一部分发送的。这样的请求头的名称是“cookie”，其对应值是一些名/值对。下面是一个 Cookie 请求头的例子，其中包含两个 Cookie：userName 和 password。

```
Cookie: userName=budi; password=pwd;
```

对 Cookie 的解析是通过 org.apache.catalina.util.RequestUtil 类的 parseCookieHeader() 方法完成的。该方法接受 Cookie 请求头，返回 javax.servlet.http.Cookie 类型的一个数组。数组中元素的个数与 Cookie 请求头中名/值对的数目相同。代码清单 3-5 给出了 parseCookieHeader() 方法的实现。

代码清单 3-5 org.apache.catalina.util.RequestUtil 类的 parseCookieHeader() 方法的实现

```

public static Cookie[] parseCookieHeader(String header) {
    if ((header == null) || (header.length() < 1))
        return (new Cookie[0]);

    ArrayList cookies = new ArrayList();
    while (header.length() > 0) {
        int semicolon = header.indexOf(';');
        if (semicolon < 0)
            semicolon = header.length();
        if (semicolon == 0)
            break;
        String token = header.substring(0, semicolon);
        if (semicolon < header.length())
            header = header.substring(semicolon + 1);
        else
            header = "";
        try {
            int equals = token.indexOf('=');
            if (equals > 0) {
                String name = token.substring(0, equals).trim();
                String value = token.substring(equals+1).trim();
                cookies.add(new Cookie(name, value));
            }
        } catch (Throwable e) {
            ;
        }
    }
    return ((Cookie[]) cookies.toArray(new Cookie[cookies.size()]));
}

```

下面是 HttpProcessor 类的 parseHeader() 方法的一部分，后者处理 Cookie 的实现：



```

else if (header.equals(DefaultHeaders.COOKIE_NAME)) {
    Cookie cookies[] = RequestUtil.parseCookieHeader(value);
    for (int i = 0; i < cookies.length; i++) {
        if (cookies[i].getName().equals("jsessionid")) {
            // Override anything requested in the URL
            if (!request.isRequestedSessionIdFromCookie()) {
                // Accept only the first session id cookie
                request.setRequestedSessionId(cookies[i].getValue());
                request.setRequestedSessionCookie(true);
                request.setRequestedSessionURL(false);
            }
        }
        request.addCookie(cookies[i]);
    }
}
}

```

### 5. 获取参数

在调用 `javax.servlet.http.HttpServletRequest` 的 `getParameter()`、`getParameterMap()`、`getParameterNames()` 或 `getParameterValues()` 方法之前，都不需要解析查询字符串或 HTTP 请求体来获得参数。因此，在 `HttpRequest` 类中，这 4 个方法的实现都会先调用 `parseParameter()` 方法。

参数只需要解析一次即可，而且也只会解析一次，因为，在请求体中包含参数，解析参数的工作会使 `SocketInputStream` 类读完整个字节流。`HttpRequest` 类使用一个名为 `parsed` 的布尔变量来标识是否已经完成对参数的解析。

参数可以出现在查询字符串或请求体中。若用户使用 GET 方法请求 servlet，则所有的参数都会在查询字符串中；若用户使用 POST 方法请求 servlet，则请求体中也可能会有参数。所有的名/值对都会存储在一个 `HashMap` 对象中。servlet 程序员可以将其作为一个 `Map` 对象获取参数（通过调用 `HttpServletRequest` 类的 `getParameterMap()` 方法）和参数名/值。但有一点需要注意，servlet 程序员不可以修改参数值。因此，这里使用了一个特殊的 `HashMap` 类：`org.apache.catalina.util.ParameterMap`。

`ParameterMap` 类继承自 `java.util.HashMap`，其中有一个名为 `locked` 的布尔变量。只有当变量 `locked` 值为 `false` 时，才可以对 `ParameterMap` 对象中的名/值对进行添加、更新或者删除操作。否则，会抛出 `IllegalStateException` 异常。然而，可以在任何时候读取该值。`ParameterMap` 类的定义在代码清单 3-6 中给出。它重写了对值进行添加、更新和删除的方法。对值的读取可在任何时候执行，但只有当变量 `locked` 的值为 `false` 时，才能调用添加、更新和删除值的方法。

代码清单 3-6 `org.apache.Catalina.util.ParameterMap` 类

```

package org.apache.catalina.util;

import java.util.HashMap;
import java.util.Map;

public final class ParameterMap extends HashMap {
    public ParameterMap() {
        super();
    }

    public ParameterMap(int initialCapacity) {
        super(initialCapacity);
    }

    public ParameterMap(int initialCapacity, float loadFactor) {
        super(initialCapacity, loadFactor);
    }
}

```

```

    }
    public ParameterMap(Map map) {
        super(map);
    }
    private boolean locked = false;
    public boolean isLocked() {
        return (this.locked);
    }
    public void setLocked(boolean locked) {
        this.locked = locked;
    }
    private static final StringManager sm =
        StringManager.getManager("org.apache.catalina.util");
    public void clear() {
        if (locked)
            throw new IllegalStateException(
                sm.getString("parameterMap.locked"));
        super.clear();
    }
    public Object put(Object key, Object value) {
        if (locked)
            throw new IllegalStateException(
                sm.getString("parameterMap.locked"));
        return (super.put(key, value));
    }
    public void putAll(Map map) {
        if (locked)
            throw new IllegalStateException(
                sm.getString("parameterMap.locked"));
        super.putAll(map);
    }
    public Object remove(Object key) {
        if (locked)
            throw new IllegalStateException(
                sm.getString("parameterMap.locked"));
        return (super.remove(key));
    }
}

```

下面，来看一下 `parseParameters()` 方法到底是如何工作的。

由于参数可以存在于查询字符串或 HTTP 请求体中，因此 `parseParameters()` 方法必须对这两者都进行检查。当解析完成时，参数会存储到对象变量 `parameters` 中，所以 `parseParameters()` 方法首先会检查布尔变量 `parsed`，若该变量值为 `true`，则方法直接返回：

```

if (parsed)
    return;

```

然后，`parseParameters()` 方法会创建一个名为 `results` 的 `ParameterMap` 类型的变量，将其指向变量 `parameters`。若变量 `parameters` 为 `null`，则 `parseParameters()` 方法会新创建一个 `ParameterMap` 对象：

```

ParameterMap results = parameters;
if (results == null)
    results = new ParameterMap();

```

然后，`parseParameters()` 方法打开 `parameterMap` 对象的锁，使其可写：

```
results.setLocked(false);
```

接下来, `parseParameters()` 方法检查字符串 `encoding`, 若 `encoding` 为 `null`, 则使用默认编码:

```
String encoding = getCharacterEncoding();
if (encoding == null)
    encoding = "ISO-8859-1";
```

然后, `parseParameters()` 方法会对参数进行解析, 解析工作会调用 `org.apache.Catalina.util.RequestUtil` 类的 `parseParameters()` 方法完成:

```
// Parse any parameters specified in the query string
String queryString = getQueryString();
try {
    RequestUtil.parseParameters(results, queryString, encoding);
}
catch (UnsupportedEncodingException e) {
    ;
}
```

接着, `parseParameters()` 方法会检查 HTTP 请求体是否包含请求参数。若用户使用 POST 方法提交请求时, 请求体会包含参数, 则请求头 “`content-length`” 的值会大于 0, “`content-type`” 的值为 “`application/x-www-form-urlencoded`”。下面的代码用于解析请求体:

```
// Parse any parameters specified in the input stream
String contentType = getContentType();
if (contentType == null)
    contentType = "";
int semicolon = contentType.indexOf(';');
if (semicolon >= 0) {
    contentType = contentType.substring(0, semicolon).trim();
}
else {
    contentType = contentType.trim();
}
if ("POST".equals(getMethod()) && (getContentLength() > 0)
    && "application/x-www-form-urlencoded".equals(contentType)) {
    try {
        int max = getContentLength();
        int len = 0;
        byte buf[] = new byte[getContentLength()];
        ServletInputStream is = getInputStream();
        while (len < max) {
            int next = is.read(buf, len, max - len);
            if (next < 0) {
                break;
            }
            len += next;
        }
        is.close();
        if (len < max) {
            throw new RuntimeException("Content length mismatch");
        }
        RequestUtil.parseParameters(results, buf, encoding);
    }
    catch (UnsupportedEncodingException ue) {
        ;
    }
    catch (IOException e) {
        throw new RuntimeException("Content read fail");
    }
}
```



最后, `parseParameters()` 方法会锁定 `ParameterMap` 对象, 将布尔变量 `parsed` 设置为 `true`, 将变量 `results` 赋值给变量 `parameters`:

```
// Store the final results
results.setLocked(true);
parsed = true;
parameters = results;
```

### 3.2.4 创建 `HttpResponse` 对象

`HttpResponse` 类实现 `javax.servlet.http.HttpServletResponse` 接口, 与其对应的外观类是 `HttpResponseFacade`。图 3-3 展示了 `HttpResponse` 类及其相关类的 UML 类图。

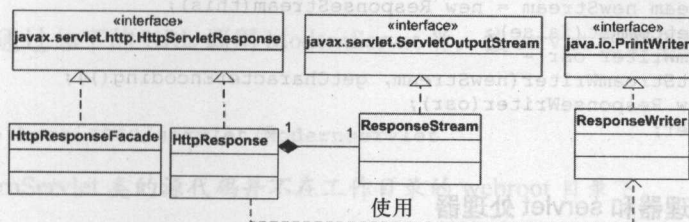


图 3-3 `HttpResponse` 及其相关类的 UML 类图

在第 2 章中, `HttpResponse` 类只实现了部分功能。例如, 当调用它的其中一个 `print()` 方法时, 它的 `getWriter()` 方法返回的 `java.io.PrintWriter` 对象并不会自动将结果发送到客户端。本章的应用程序将解决此问题。在此之前, 为了便于理解, 先说明一下什么是 `Writer`。

在 `Servlet` 中, 可以使用 `PrintWriter` 对象向输出流中写字符。可以使用任意编码格式, 但在向浏览器发送字符的时候, 实际上都是字节流。因此, 对于第 2 章, 在 `ex02.pyrmont.HttpResponse` 类中使用了如下的 `getWriter()` 方法, 也就没什么好奇怪的了:

```
public PrintWriter getWriter() {
    // if autoflush is true, println() will flush,
    // but print() will not.
    // the output argument is an OutputStream
    writer = new PrintWriter(output, true);
    return writer;
}
```

那么, 如何创建 `PrintWriter` 对象呢? 可以通过传入一个 `java.io.OutputStream` 实例来创建 `PrintWriter` 对象。所传给 `PrintWriter` 类的 `print()` 方法或 `println()` 方法的任何字符串都会被转换为字节流, 使用基本的输出流发送到客户端。

在第 3 章中, 会使用 `ex03.pyrmont.connector.ResponseStream` 类的一个实例作为 `PrintWriter` 类的输出流对象。注意, `ResponseStream` 类 `java.io.OutputStream` 类的直接子类。

也可以使用 `ex03.pyrmont.connector.ResponseWriter` 类来向客户端发送信息, 该类继承自 `PrintWriter` 类。 `ResponseWriter` 类重写了所有的 `print()` 方法和 `println()` 方法, 这样对这些方法进行调用时, 会自动将信息发送客户端。因此使用一个 `ResponseWriter` 实例和一个基本 `ResponseStream` 对象。

可以通过传入 `ResponseStream` 对象的一个实例来实例化 `ResponseWriter` 类。但是, 这里使

用了一个 `java.io.OutputStreamWriter` 对象作为 `ResponseWriter` 对象和 `ResponseStream` 对象之间的桥梁。

使用 `OutputStreamWriter` 类，传入的字符会被转换为使用指定字符集的字节数组。其中所使用的字符集可以通过名称显式指定，也可以使用平台的默认字符集。每次调用写方法时，都会先使用编码转换器对给定字符进行编码转换。在被写入基本输出流之前，返回的字节数组会先存储在缓冲区中。缓冲区的大小是固定的，对于大多数应用来说，其默认值是足够大的。注意，传递给写方法的字符是没有缓冲的。

因此，`getWriter()` 方法的实现如下所示：

```
public PrintWriter getWriter() throws IOException {
    ResponseStream newStream = new ResponseStream(this);
    newStream.setCommit(false);
    OutputStreamWriter osr =
        new OutputStreamWriter(newStream, getCharacterEncoding());
    writer = new ResponseWriter(osr);
    return writer;
}
```

### 3.2.5 静态资源处理器和 servlet 处理器

本章的 `ServletProcessor` 类与第2章的 `ex02.pyrmont.ServletProcessor` 类相似，都只有一个方法：`process()` 方法。但是，`ex03.pyrmont.connector.ServletProcessor` 类中的 `process()` 方法会接受一个 `HttpRequest` 对象和一个 `HttpResponse` 对象，而不是 `Request` 或 `Response` 的实例。下面是本章应用程序中 `process()` 方法的签名：

```
public void process(HttpRequest request, HttpResponse response);
```

此外，`process()` 方法使用 `HttpRequestFacade` 类和 `HttpResponseFacade` 类作为 `request` 和 `response` 对象的外观类。当调用完 `servlet` 的 `service()` 方法后，它还会调用一次 `HttpResponse` 类的 `finishResponse()` 方法：

```
servlet = (Servlet) myClass.newInstance();
HttpRequestFacade requestFacade = new HttpRequestFacade(request);
HttpResponseFacade responseFacade = new
    HttpResponseFacade(response);
servlet.service(requestFacade, responseFacade);
((HttpResponse) response).finishResponse();
```

本章应用程序中的 `StaticResourceProcessor` 类几乎与 `ex02.pyrmont.StaticResourceProcessor` 类完全相同。

### 3.2.6 运行应用程序

要在 Windows 平台下运行该应用程序，需要在工作目录下执行如下命令：

```
java -classpath ./lib/servlet.jar;./ ex03.pyrmont.startup.Bootstrap
```

要在 Linux 平台下运行，需要使用冒号来代替库文件之间的分号：

```
java -classpath ./lib/servlet.jar:./ ex03.pyrmont.startup.Bootstrap
```

要显示 index.html 文件，可以使用如下的 URL：

```
http://localhost:8080/index.html
```

要调用 Primitiveservlet 类，可以在浏览器中使用如下的 URL：

```
http://localhost:8080/servlet/Primitiveservlet
```

可以在浏览器中看到如下的输入：

```
Hello. Roses are red.
```

```
Violets are blue.
```

**注意** 在第 2 章运行 PrimitiveServlet 不会显示第 2 行内容。

现在，可以通过如下的 URL 调用 ModernServlet 类，该 servlet 在第 2 章的 servlet 容器中是无法运行的：

```
http://localhost:8080/servlet/Modernservlet
```

**注意** ModernServlet 类的源代码并不在工作目录的 webroot 目录下。

可以在测试 servlet 的 URL 后面添加查询字符串。例如使用如下的 URL：

```
http://localhost:8080/servlet/ModernServlet?userName=tarzan&password=pwd
```

图 3-4 给出了使用上述 URL 运行 ModernServlet 的返回结果：

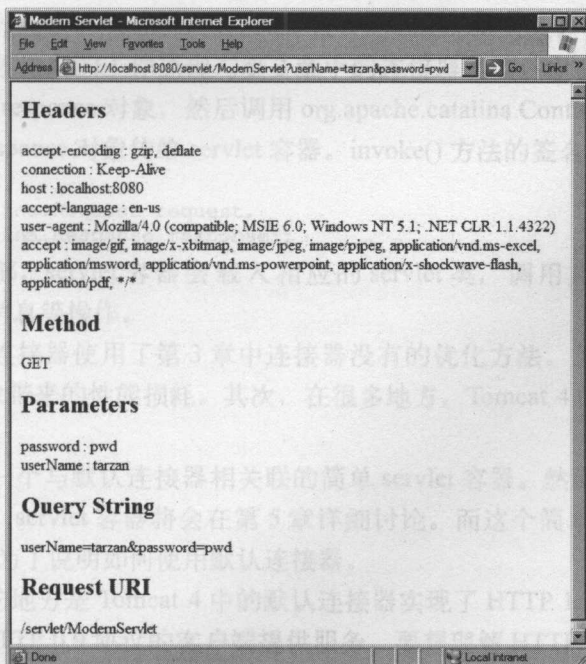


图 3-4 运行 ModernServlet 的返回结果



### 3.3 小结

本章学习了连接器是如何工作的。本章所构建的连接器是 Tomcat 4 中默认连接器的一个简化版。如你所知, 由于该默认连接器性能不高, 已经不推荐使用。例如, 所有的 HTTP 请求头都会被解析, 即使并不会在 servlet 中使用它们。因此, 默认连接器运行缓慢, 现在已经被 Coyote 连接器所替代。Coyote 连接器执行起来速度更快一些, 其源代码可以在 Apache 软件基金会的网站上下载。但无论如何, 默认连接器是一个不错的学习工具, 将在第 4 章详细讨论。

因此, `getWhenReady()` 方法的实现如下所示:

```
public PrintWriter testWhenReady() throws IOException {
    PrintWriter writer = new PrintWriter(new OutputStreamWriter(
        new FileOutputStream("testWhenReady.txt"), "UTF-8"), "UTF-8");
    writer.println("testWhenReady");
    writer.close();
}
```

#### 3.2.5 静态资源处理和 Servlet 处理

在 Tomcat 4 中, 静态资源处理和 Servlet 处理是分开的。静态资源处理由 `StaticResourceProcessor` 类负责, 而 Servlet 处理则由 `ServletProcessor` 类负责。在 `ServletProcessor` 类中, 有一个 `process()` 方法, 用于处理 Servlet 请求。该方法会调用 `Servlet` 接口的 `service()` 方法, 从而将请求转发给相应的 Servlet 实例。

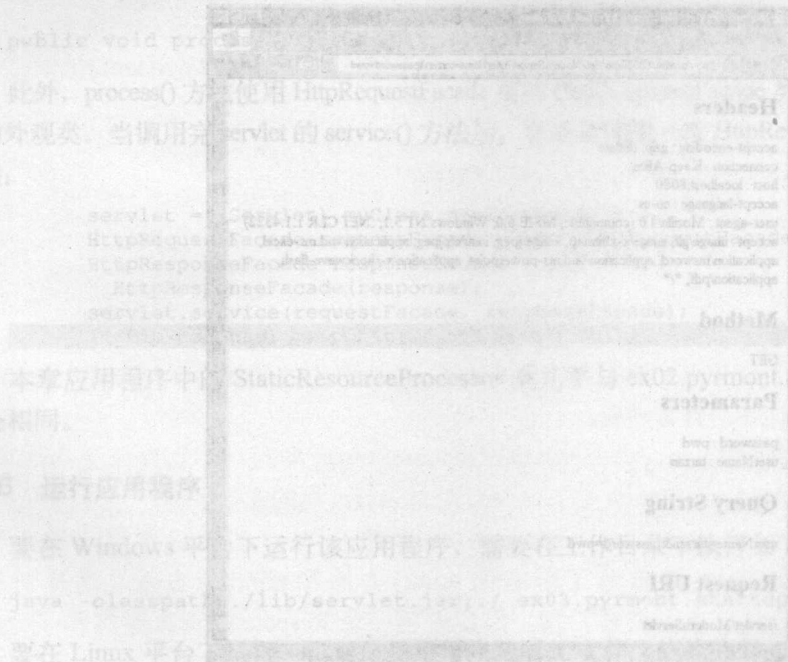


图 3-4 运行 `ServletProcessor` 的返回结果

## 第 ④ 章

# Tomcat 的默认连接器

第 3 章中的连接器已经可以正常地工作，经过修改后还可以实现更多的功能。但是，它终究只是一个学习工具，是为了介绍 Tomcat 的默认连接器而写。理解第 3 章中编写的连接器的工作原理是理解本章中介绍的 Tomcat 4 中的默认连接器的关键。在第 4 章中，通过对 Tomcat 4 中默认连接器的源代码的剖析来讨论如何构建一个真正的 Tomcat 连接器。

**注意** 本章中的“默认连接器”指的是 Tomcat 4 中的默认连接器。虽然该连接器已经弃用，被另一个运行速度更快的连接器——Coyote——取代，但它仍然是一个不错的学习工具。

Tomcat 中的连接器是一个独立的模块，可以被插入到 servlet 容器中，而且还有很多连接器可以使用。例如 Coyote、mod\_jk、mod\_jk2 和 mod\_webapp 等。Tomcat 中使用的连接器必须满足以下要求：

- 实现 org.apache.catalina.Connector 接口；
- 负责创建实现了 org.apache.catalina.Request 接口的 request 对象；
- 负责创建实现了 org.apache.catalina.Response 接口的 response 对象。

Tomcat 4 中的默认连接器的工作原理与第 3 章中的连接器类似。它会等待引入的 HTTP 请求，创建 request 对象和 response 对象，然后调用 org.apache.catalina.Container 接口的 invoke() 方法，将 request 对象和 response 对象传给 servlet 容器。invoke() 方法的签名如下：

```
public void invoke(  
    org.apache.catalina.Request request,  
    org.apache.catalina.Response response);
```

在 invoke() 方法内部，servlet 容器会载入相应的 servlet 类，调用其 service() 方法，管理 session 对象，记录错误消息等操作。

Tomcat 4 中的默认连接器使用了第 3 章中连接器没有的优化方法。首先，使用了一个对象池来避免了频繁创建对象带来的性能损耗。其次，在很多地方，Tomcat 4 的默认连接器使用了字符数组来代替字符串。

本章的应用程序是一个与默认连接器相关联的简单 servlet 容器。然而，本章关注默认连接器而不是简单的连接器。servlet 容器将会在第 5 章详细讨论。而这个简单的 servlet 容器将会在第 4.9 节进行讨论，主要是为了说明如何使用默认连接器。

另外一个需要注意的地方是 Tomcat 4 中的默认连接器实现了 HTTP 1.1 的全部新特性，也可以为使用 HTTP 1.0 和 HTTP 0.9 协议的客户端提供服务。要想理解 HTTP 1.1 中的新特性，你应该认真读一下 4.2 节中的内容。然后，我们会对 org.apache.catalina.Connector 接口进行说明，讨论其是如何创建 request 对象和 response 对象的。如果你已经理解第 3 章中描述的连接器是如何

工作的，那么在理解默认连接器的工作原理时也不会有任何问题。

本章会先从介绍 HTTP 1.1 的 3 个新特性开始。这是理解 Tomcat 4 中默认连接器内部工作原理的关键点。然后，会介绍 `org.apache.catalina.Connector` 接口，Tomcat 中所有的连接器都必须实现该接口。本章会使用第 3 章的一些应用程序中已经出现的类，如 `HttpConnector` 和 `HttpProcessor` 等类，但是，这些类都比原先复杂了一些。

## 4.1 HTTP 1.1 的新特性

本节会介绍 HTTP 1.1 中的 3 个新特性。理解这些新特性对理解默认连接器如何处理 HTTP 请求至关重要。

### 4.1.1 持久连接

在 HTTP 1.1 之前，无论浏览器何时连接到 Web 服务器，当服务器将请求的资源返回后，就会断开与浏览器的连接。但是，网页上会包含一些其他资源，如图片文件、applet 等。因此，当请求一个页面时，浏览器还需要下载这些被页面引用的资源。如果页面和它引用的所有资源文件都使用不同的连接进行下载的话，处理过程会很慢。这就是为什么 HTTP 1.1 中会引入持久连接。使用持久连接后，当下载了页面后，服务器并不会立即关闭连接。相反，它会等待 Web 客户端请求被该页面所引用的所有资源。这样一来，页面和被页面引用的资源都会使用同一个连接来下载。考虑到建立 / 关闭 HTTP 连接是一个系统开销很大的操作，使用同一个连接来下载所有的资源会为 Web 服务器、客户端和网络节省很多时间和工作量。

在 HTTP 1.1 中，会默认使用持久连接。当然，也可以显式地使用，方法是浏览器发送如下请求头信息：

```
connection: keep-alive
```

### 4.1.2 块编码

建立了持久连接后，服务器可以从多个资源发送字节流，而客户端也可以使用该连接发送多个请求。这样的结果就是发送方必须在每个请求或响应中添加“content-length”头信息，这样，接收方才能知道如何解释这些字节信息。但通常情况下，发送方并不知道要发送多少字节。例如，servlet 容器可能要在接收到一些字节之后，就开始发送响应信息，而不必等到接收完所有的信息。这就是说，必须有一种方法来告诉接收方在不知道发送内容长度的情况下，如何解析已经接收到的内容。

其实，即使没有发出多个请求或发送多个响应，服务器或客户端也不需要知道有多少字节要发送。在 HTTP 1.0 中，服务器可以不写“content-length”头信息，尽管往连接中写响应内容就行了。当发送完响应信息后，它就直接关闭连接。在这种情况下，客户端会一直读取内容，直到读方法返回 -1，这表明已经读到了文件末尾。

HTTP 1.1 使用一个名为“transfer-encoding”的特殊请求头，来指明字节流将会分块发送。对每一个块，块的长度（以十六进制表示）后面会有一个回车 / 换行符（CR/LF），然后是具体



的数据。一个事务以一个长度为 0 的块标记。假设要用两个块发送下面 38 个字节的内容，其中第 1 个块为 29 个字节，第 2 个块为 9 个字节：

```
I'm as helpless as a kitten up a tree.
```

那么，实际上应该发送如下内容：

```
1D\r\n
I'm as helpless as a kitten u
9\r\n
p a tree.
0\r\n
```

“1D”的十进制表示是 29，表明第 1 个块的长度是 29 个字节，“0\r\n”表明事务已经完成。

#### 4.1.3 状态码 100 的使用

使用 HTTP 1.1 的客户端可以在向服务器发送请求体之前发送如下的请求头，并等待服务器的确认：

```
Expect: 100-continue
```

当客户端准备发送一个较长的请求体，而不确定服务端是否会接收时，就可能会发送上面的头信息。若是客户端发送了较长的请求体，却发现服务器拒绝接收时，会是较大的浪费。

接收到“Expect: 100-continue”请求头后，若服务器可以接收并处理该请求时，可以发送如下的响应头：

```
HTTP/1.1 100 Continue
```

注意，返回内容后面要加上 CRLF 字符。

然后，服务器继续读取输入流的内容。

## 4.2 Connector 接口

Tomcat 的连接器必须实现 `org.apache.catalina.Connector` 接口。在接口中声明了很多方法，其中最重要的是 `getContainer()`、`setContainer()`、`createRequest()` 和 `createResponse()`。

`setContainer()` 方法用于将连接器和某个 servlet 容器相关联。`getContainer()` 方法返回与当前连接器相关联的 servlet 容器。`createRequest()` 方法会为引入的 HTTP 请求创建 request 对象，相应地，`createResponse()` 方法会创建一个 response 对象。

`org.apache.catalina.connector.http.HttpConnector` 类实现了 `Connector` 接口，该类将会在 4.4 节中进行讨论。现在先看一下图 4-1 中默认连接器的 UML 类图。要注意的是，为了使类图简单一些，Request 和 Response 接口的实现类已经省略掉了。除了 `SimpleContainer` 类之外，其他类或接口的 `org.apache.catalina` 包名前缀也省略掉了。因此 `Connector` 接口的完全限定名实际上是 `org.apache.catalina.Connector`，`Util.StringManager` 类的完全限定名实际上是 `org.apache.catalina.util.StringManager`。

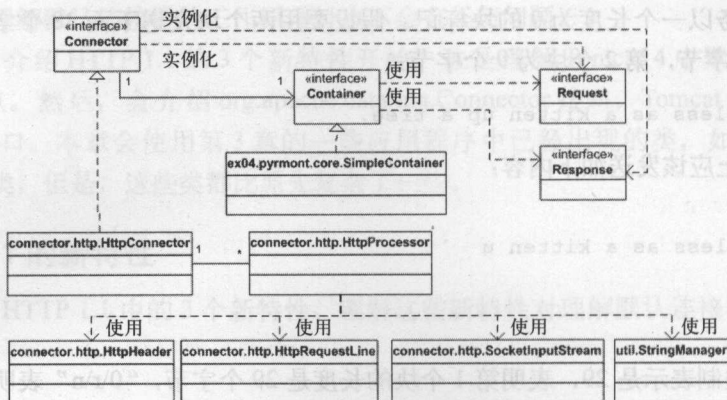


图 4-1 默认连接器的 UML 类图

连接器与 servlet 容器是一对一的关系，箭头指向表示的关系是连接器知道要用哪个 servlet 容器。此外，与第 3 章不同，`HttpConnector` 类与 `HttpProcessor` 类是一对多的关系。

### 4.3 `HttpConnector` 类

在第 3 章中，已经实现了一个与 `org.apache.catalina.connector.http.HttpConnector` 类似的简化版连接器。它实现了 `org.apache.catalina.Connector` 接口（使其可以成为 Catalina 中的连接器）、`java.lang.Runnable` 接口（确保它的实例在自己的线程中运行）和 `org.apache.catalina.Lifecycle` 接口。Lifecycle 接口用于维护每个实现了该接口的每个 Catalina 组件的生命周期。

Lifecycle 接口的具体内容将在第 6 章介绍。现在你只需要知道，由于 `HttpConnector` 类实现了 Lifecycle 接口，因此当创建一个 `HttpConnector` 实例后，就应该调用其 `initialize()` 方法和 `start()` 方法。在组件的整个生命周期内，这两个方法只应该被调用一次。下面要介绍一些与第 3 章中的 `HttpConnector` 类不同的功能：`HttpConnector` 如何创建服务器套接字，它如何维护 `HttpProcessor` 池，以及它如何提供 Http 请求服务。

#### 4.3.1 创建服务器套接字

`HttpConnector` 类的 `initialize()` 方法会调用一个私有方法 `open()`，后者返回一个 `java.net.ServerSocket` 实例，赋值给成员变量 `serverSocket`。但是，这里并没有直接调用 `ServerSocket` 类的构造函数，而是通过 `open()` 方法从一个服务器套接字工厂得到一个实例。若你了解该工厂方法的具体实现，可以阅读参考 `ServerSocketFactory` 类和 `DefaultServerSocketFactory` 类（都在 `org.apache.catalina.net` 包内）的源代码。

#### 4.3.2 维护 `HttpProcessor` 实例

在第 3 章的应用程序中，每次 `HttpConnector` 实例只有一个 `HttpProcessor` 实例可以使用，所有，每次它只能处理一个 HTTP 请求。在 Tomcat 的默认连接器中，`HttpConnector` 实例有一个 `HttpProcessor` 对象池，每个 `HttpProcessor` 实例都运行在其自己的线程中。这样，`HttpConnector`

实例就可以同时处理多个 HTTP 请求了。

HttpConnector 实例维护一个 HttpProcessor 实例池，避免每次都为新请求创建 HttpProcessor 对象的操作。HttpProcessor 实例存储在一个名为 processors 的 java.io.Stack 类型变量中：

```
private Stack processors = new Stack();
```

在 HttpConnector 中，创建的 HttpProcessor 实例的个数由两个变量决定：minProcessors 和 maxProcessors。默认情况下，minProcessors 的值为 5，maxProcessors 的值为 20，可以通过 setMinProcessors() 方法和 setMaxProcessors() 方法对这两个数值进行修改：

```
protected int minProcessors = 5;  
private int maxProcessors = 20;
```

初始，HttpConnector 对象会依据 minProcessors 的数值来创建 HttpProcessor 实例。若是请求的数目超过了 HttpProcessor 实例所能处理的范围，HttpConnector 实例就会创建更多的 HttpProcessor 实例，直到 HttpProcessor 实例的数目达到 maxProcessors 限定的范围。若是 HttpProcessor 实例的数目已经达到了 maxProcessors 限定的数值，但还是不够用，此时引入的 HTTP 请求就会被忽略掉。若希望 HttpConnector 可以持续地创建 HttpProcessor 实例，就可以将变量 maxProcessors 的值设置为负数。此外，变量 curProcessors 保存了当前已有的 HttpProcessor 实例的数量。

下面的代码是在 HttpConnector 类中的 start() 方法中创建初始数量的 HttpProcessor 实例的部分实现：

```
while (curProcessors < minProcessors) {  
    if ((maxProcessors > 0) && (curProcessors >= maxProcessors))  
        break;  
    HttpProcessor processor = newProcessor();  
    recycle(processor);  
}
```

其中 newProcessor() 方法负责创建 HttpProcessor 实例，并将 curProcessors 加 1。recycle() 方法将新创建的 HttpProcessor 对象入栈。

每个 HttpProcessor 实例负责解析 HTTP 请求行和请求头，填充 request 对象。因此，每个 HttpProcessor 对象都关联一个 request 对象和一个 response 对象。HttpProcessor 类的构造函数会调用 HttpConnector 类的 createRequest() 方法和 createResponse() 方法。

### 4.3.3 提供 HTTP 请求服务

HttpConnector 类的主要业务逻辑在其 run() 方法中（就像在第 3 章的应用程序中那样）。run() 方法包含一个 while 循环，在该循环体内，服务器套接字等待 HTTP 请求，直到 HttpConnector 对象关闭。

```
while (!stopped) {  
    Socket socket = null;  
    try {  
        socket = serverSocket.accept();  
        ...  
    }
```



对于每个引入的 HTTP 请求，它通过调用其私有方法 `createProcessor()` 获得一个 `HttpProcessor` 对象。

```
HttpProcessor processor = createProcessor();
```

但是，大多数时间里，`createProcessor()` 方法并不会创建一个新的 `HttpProcessor` 实例。相反，它会从池中获取一个对象。如果栈中还有一个 `HttpProcessor` 实例可以使用，`createProcessor()` 方法就从栈中弹出一个 `HttpProcessor` 实例，将其返回。如果栈已经空了，而已经创建的 `HttpProcessor` 实例的数量还没有超过限定的最大值，`createProcessor()` 方法就会创建一个新的 `HttpProcessor` 实例。但是，若 `HttpProcessor` 实例的数量已经达到了限定的最大值，`createProcessor()` 方法返回 `null`。此时，服务器会简单地关闭套接字，不对这个引入的 HTTP 请求进行处理：

```
if (processor == null) {  
    try {  
        log(sm.getString("httpConnector.noProcessor"));  
        socket.close();  
    }  
    ...  
    continue;  
}
```

若 `createProcessor()` 方法的返回值不为 `null`，则会将客户端套接字传入到 `HttpProcessor` 类的 `assign()` 方法中：

```
processor.assign(socket);
```

现在，`HttpProcessor` 实例的任务是读取套接字的输入流，解析 HTTP 请求。这里需要注意的是，`assign()` 方法直接返回，而不要等待 `HttpProcessor` 实例完成解析，这样 `HttpConnector` 才能接续服务传入的 HTTP 请求，而 `HttpProcessor` 实例是在其自己的线程中完成解析工作的，具体内容参见 4.4 节。

## 4.4 HttpProcessor 类

默认连接器中的 `HttpProcessor` 类是第 3 章中同名类的全功能版本。你已经了解了其工作原理，本章将着重于讲解 `HttpProcessor` 类中 `assign()` 方法的异步实现，这样 `HttpConnector` 实例就能同时处理多个 HTTP 请求。

**注意** `HttpProcessor` 类中另一个重要的方法是其私有方法 `process()` 方法，该方法负责解析 HTTP 请求，并调用相应的 `Servlet` 容器的 `invoke` 方法。4.7 节将对其进行讲解。

在第 3 章中，`HttpConnector` 实例是运行在其自己的线程中的。但是，它必须等待当前正在处理的 HTTP 请求的返回，然后才能处理下一个 HTTP 请求。下面是第 3 章中 `HttpConnector` 类的 `run()` 方法的部分代码：

```
public void run() {  
    ...  
    while (!stopped) {  
        Socket socket = null;  
        try {  
            socket = serverSocket.accept();
```

```
}  
catch (Exception e) {  
    continue;  
}  
// Hand this socket off to an HttpProcessor  
HttpProcessor processor = new HttpProcessor(this);  
processor.process(socket);  
}  
}
```

第3章中 `HttpProcessor` 类的 `process()` 方法是同步的，因此，在接受下一个 HTTP 请求前，`run()` 方法会一直等待，直到 `process()` 方法返回。

但是，在 Tomcat 的默认连接器中，`HttpProcessor` 类实现了 `java.lang.Runnable` 接口，这样，每个 `HttpProcessor` 实例就可以运行在自己的线程中了，称为“处理器线程”。为每个 `HttpConnector` 对象创建 `HttpProcessor` 实例后，会调用其 `start()` 方法，启动 `HttpProcessor` 实例的处理器线程。代码清单 4-1 展示了默认连接器中 `HttpProcessor` 类的 `run()` 方法的实现。

代码清单 4-1 `HttpProcessor` 类中 `run()` 方法的实现

```
public void run() {  
    // Process requests until we receive a shutdown signal  
    while (!stopped) {  
        // Wait for the next socket to be assigned  
        Socket socket = await();  
        if (socket == null)  
            continue;  
        // Process the request from this socket  
        try {  
            process(socket);  
        }  
        catch (Throwable t) {  
            log("process.invoke", t);  
        }  
        // Finish up this request  
        connector.recycle(this);  
    }  
    // Tell threadStop() we have shut ourselves down successfully  
    synchronized (threadSync) {  
        threadSync.notifyAll();  
    }  
}
```

`run()` 方法中的 `while` 循环会依次做如下几件事：获取套接字对象，进行处理，调用连接器的 `recycle()` 方法将当前的 `HttpProcessor` 实例压回栈中。下面是 `HttpConnector` 类中 `recycle()` 方法的实现代码：

```
void recycle(HttpProcessor processor) {  
    processors.push(processor);  
}
```

注意，`run()` 方法中的 `while` 循环在执行到 `await()` 方法时会阻塞。`await()` 方法会阻塞处理器线程的控制流，直到它从 `HttpConnector` 中获取到了新的 `Socket` 对象。换句话说，就是直到 `HttpConnector` 对象调用 `HttpProcessor` 实例的 `assign()` 方法前，都会一直阻塞。但是，`await()` 方

法与 assign() 方法并不是运行在同一个线程中的。assign() 方法是从 HttpURLConnection 对象的 run() 方法中调用的。我们称 HttpURLConnection 实例中 run() 方法运行时所在的线程为“连接器线程”。那么 assign() 方法如何通知 await() 方法，自己已经被调用了呢？通过使用一个名为 available 的布尔变量和 java.lang.Object 类的 wait() 方法和 notifyAll() 方法来进行沟通的。

**注意** wait() 方法会使当前线程进入等待状态，直到其他线程调用了这个对象的 notify() 方法和 notifyAll() 方法。

下面的代码是 HttpProcessor 类的 assign() 方法和 await() 方法的实现：

```
synchronized void assign(Socket socket) {
    // Wait for the Processor to get the previous Socket
    while (available) {
        try {
            wait();
        }
        catch (InterruptedException e) {
        }
    }
    // Store the newly available Socket and notify our thread
    this.socket = socket;
    available = true;
    notifyAll();
    ...
}

private synchronized Socket await() {
    // Wait for the Connector to provide a new Socket
    while (!available) {
        try {
            wait();
        }
        catch (InterruptedException e) {
        }
    }

    // Notify the Connector that we have received this Socket
    Socket socket = this.socket;
    available = false;
    notifyAll();
    if ((debug >= 1) && (socket != null))
        log(" The incoming request has been awaited");
    return (socket);
}
```

表 4-1 总结了 await() 方法和 assign() 方法的程序流。

表 4-1 assign() 方法和 await() 方法的总结

处理器线程 (await() 方法)	连接器线程 (assign() 方法)
while (!available) {	while (available) {
wait();	wait();
}	}
Socket socket = this.socket;	this.socket = socket;
available = false;	available = true;
notifyAll();	notifyAll();
return socket; // to the run	...
// method	



当“处理器线程”刚刚启动时，available 变量的值为 false，所以线程会在 while 循环内一直等待（参见表 4-1 的第 1 列）。它会等待到其他线程调用了 notify() 方法或 notifyAll() 方法。也就是说，调用 wait() 方法会使用“处理器线程”暂停，直到“连接器线程”调用了 HttpProcessor 实例的 notifyAll() 方法。

然后，看第 2 列，当一个新的 Socket 对象通过 assign() 方法传入时，“连接器线程”会调用 HttpProcessor 实例的 assign() 方法。由于变量 available 的值为 false，因此会跳过 while 循环，传入的 Socket 对象会赋值给 HttpProcessor 实例的 socket 变量：

```
this.socket = socket;
```

然后，“连接器线程”会将变量 available 的值设置为 true，并调用 notifyAll() 方法。这会唤醒“处理器线程”。由于变量 available 的值为 true，因此所有程序会跳出 while 循环，将成员变量 socket 的值赋值给一个局部变量，将变量 available 设置为 false，调用 notifyAll() 方法，然后将这个局部变量 socket 返回，由其他程序进行处理。

为什么 await() 方法要使用一个局部变量 socket，而不直接将成员变量 socket 返回呢？因为使用局部变量可以在当前 Socket 对象处理完之前，继续接收下一个 Socket 对象。

为什么 await() 方法需要调用 notifyAll() 方法？是为了防止出现另一个 Socket 对象已经到达，而此时变量 available 的值还是 true 的情况。在这种情况下，“连接器线程”会在 assign() 方法内的循环中阻塞，直到“处理器线程”调用了 notifyAll() 方法。

## 4.5 Request 对象

默认连接器中的 HTTP Request 对象是 org.apache.catalina.Request 接口的实例。RequestBase 类直接实现自该接口，而 RequestBase 类是 HttpRequest 类的父类。最终的实现类是 HttpRequestImpl 类，而 HttpRequestImpl 类继承自 HttpRequest 类。与第 3 章类似，这些类都有各自的外观类，RequestFacade 类和 HttpRequestFacade 类。图 4-2 展示了 Request 接口及其实现类的 UML 类图。注意，除了 javax.servlet 和 javax.servlet.http 包下的类外，完全限定名中的 org.apache.catalina 省略掉了。

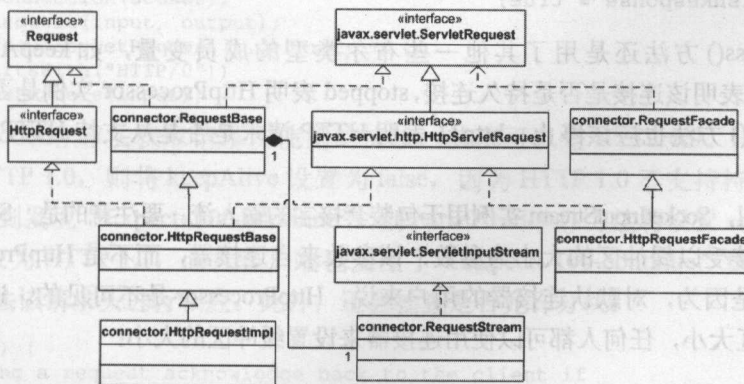


图 4-2 Request 接口及相关实现类的 UML 类图



```

SocketInputStream input = null;
OutputStream output = null;
// Construct and initialize the objects we will need
try {
    input = new SocketInputStream(socket.getInputStream(),
        connector.getBufferSize());
}
catch (Exception e) {
    ok = false;
}

```

然后是一个 while 循环，在该循环体内，不断地读取输入流，直到 `HttpProcessor` 实例终止，抛出一个异常，或连接断开：

```

keepAlive = true;
while (!stopped && ok && keepAlive) {
    ...
}

```

在 while 循环内，`process()` 方法会先将变量 `finishResponse` 的值设置为 `true`，然后获取输出流对象，执行 `request` 和 `response` 对象的一些初始化操作：

```

finishResponse = true;
try {
    request.setStream(input);
    request.setResponse(response);
    output = socket.getOutputStream();
    response.setStream(output);
    response.setRequest(request);
    ((HttpServletResponse) response.getResponse()).setHeader(
        "Server", SERVER_INFO);
}
catch (Exception e) {
    log("process.create", e); //logging is discussed in Chapter 7
    ok = false;
}

```

然后，`process()` 方法调用 `parseConnection()`、`parseRequest()` 和 `parseHeaders()` 方法开始解析引入的 HTTP 请求，这 3 个方法会在后面几节讨论：

```

try {
    if (ok) {
        parseConnection(socket);
        parseRequest(input, output);
        if (!request.getRequest().getProtocol()
            .startsWith("HTTP/0"))
            parseHeaders(input);
    }
}

```

`parseConnection()` 方法获取请求所使用的协议，其值可以是 HTTP 0.9、HTTP 1.0 或 HTTP 1.1。若值为 HTTP 1.0，则将 `keepAlive` 设置为 `false`，因为 HTTP 1.0 不支持持久连接。若是在 http 请求头中找到发现 “Expect: 100-continue”，则 `parseHeaders()` 方法将设置 `sendAck` 为 `true`。

若请求协议为 HTTP 1.1，而且客户端也发出了 “Expect: 100-continue” 请求头，会对调用 `ackRequest()` 方法该请求头进行相应。此外，还会检查是否允许分块：

```

if (http11) {
    // Sending a request acknowledge back to the client if
    // requested.
    ackRequest(output);
}

```



```
// If the protocol is HTTP/1.1, chunking is allowed.
if (connector.isChunkingAllowed())
    response.setAllowChunking(true);
}
```

ackRequest() 方法检查 sendAck 的值, 若其值为 true, 则发送下面格式的字符串:

```
HTTP/1.1 100 Continue\r\n\r\n
```

在解析 HTTP 请求的过程中, 有可能会抛出很多种异常。发生任何一个异常都会将变量 ok 或 finishResponse 设置为 false。完成解析后, process() 方法将 request 和 response 对象作为参数传入 servlet 容器的 invoke() 方法:

```
try {
    ((HttpServletResponse) response).setHeader
        ("Date", FastDateFormat.getCurrentDate());
    if (ok) {
        connector.getContainer().invoke(request, response);
    }
}
```

然后, 若变量 finishResponse 的值为 true, 则调用 response 对象的 finishResponse() 方法和 request 对象的 finishRequest() 方法, 再将结果发送至客户端:

```
if (finishResponse) {
    ...
    response.finishResponse();
    ...
    request.finishRequest();
    ...
    output.flush();
}
```

while 循环的最后一部分检查 response 的头信息 “Connection” 是否在 servlet 中被设置为 “close”, 或者协议是否是 HTTP 1.0。若这两种情况中任意一种为真, 则将 keepAlive 设置为 false。最后将 request 对象和 response 对象回收:

```
if ( "close".equals(response.getHeader("Connection")) ) {
    keepAlive = false;
}
// End of request processing
status = Constants.PROCESSOR_IDLE;
// Recycling the request and the response objects
request.recycle();
response.recycle();
}
```

在这种情况下, 若 keepAlive 为 true, 或在前面的解析工作中没有发生错误, 或 HttpProcessor 实例没有被终止, 则 while 循环则继续运行。否则调用 shutdownInput() 方法, 并关闭套接字:

```
try {
    shutdownInput(input);
    socket.close();
}
...
```

shutdownInput() 方法会检查是否有未读完的字节, 若有, 则它跳过这些字节。

### 4.7.1 解析连接

`parseConnection()` 方法会从套接字中获取 Internet 地址，将其赋值给 `HttpRequestImpl` 对象。此外，它还要检查是否使用了代理，将 `Socket` 对象赋值给 `request` 对象。代码清单 4-2 给出了 `parseConnection()` 方法的实现。

代码清单 4-2 `parseConnection()` 方法的实现

```
private void parseConnection(Socket socket)
    throws IOException, ServletException {
    if (debug >= 2)
        log(" parseConnection: address=" + socket.getInetAddress() +
            ", port=" + connector.getPort());
    ((HttpRequestImpl) request).setInet(socket.getInetAddress());
    if (proxyPort != 0)
        request.setServerPort(proxyPort);
    else
        request.setServerPort(serverPort);
    request.setSocket(socket);
}
```

### 4.7.2 解析请求

`parseRequest()` 方法实际上是第 3 章中相似方法的全功能版本。如果你已经理解了第 3 章的内容，通过阅读代码，你也可以很容易理解 `parseRequest()` 方法是如何工作的。

### 4.7.3 解析请求头

默认连接器中的 `parseHeaders()` 方法使用 `org.apache.catalina.connector.http` 包内的 `HttpHeader` 类和 `DefaultHeader` 类。`HttpHeader` 类表示一个 HTTP 请求头。这里与第 3 章不同的是，`HttpHeader` 类并没有使用字符串，而是使用了字符数组来避免代价高昂的字符串操作。`DefaultHeaders` 类是一个 `final` 类，包含了字符数组形式的标准 HTTP 请求头：

```
static final char[] AUTHORIZATION_NAME =
    "authorization".toCharArray();
static final char[] ACCEPT_LANGUAGE_NAME =
    "accept-language".toCharArray();
static final char[] COOKIE_NAME = "cookie".toCharArray();
...
```

`parseHeaders()` 方法使用 `while` 循环读取所有的 HTTP 请求信息。`while` 循环以调用 `request` 对象的 `allocateHeader()` 方法获取一个内容为空的 `HttpHeader` 实例开始。然后，该 `HttpHeader` 实例被传入 `SocketInputStream` 实例的 `readHeader()` 方法中：

```
HttpHeader header = request.allocateHeader();

// Read the next header
input.readHeader(header);
```

若所有的请求头都已经读取过了，则 `readHeader()` 方法不会再给 `HttpHeader` 实例设置 `name` 属性了。这时就可退出 `parseHeaders()` 方法了：

```

if (header.nameEnd == 0) {
    if (header.valueEnd == 0) {
        return;
    }
    else {
        throw new ServletException
            (sm.getString("httpProcessor.parseHeaders.colon"));
    }
}

```

如果一个 `HttpHeader` 有 `name`, 那么它肯定也会有 `value`:

```
String value = new String(header.value, 0, header.valueEnd);
```

接下来, 与第3章类似, `parseHeaders()` 方法将读取到的请求头的 `name` 属性的值与 `DefaultHeaders` 中的标准名称比较。注意, 这里的比较是字符数组的比较, 而不是字符串的比较:

```

if (header.equals(DefaultHeaders.AUTHORIZATION_NAME)) {
    request.setAuthorization(value);
}
else if (header.equals(DefaultHeaders.ACCEPT_LANGUAGE_NAME)) {
    parseAcceptLanguage(value);
}
else if (header.equals(DefaultHeaders.COOKIE_NAME)) {
    // parse cookie
}
else if (header.equals(DefaultHeaders.CONTENT_LENGTH_NAME)) {
    // get content length
}
else if (header.equals(DefaultHeaders.CONTENT_TYPE_NAME)) {
    request.setContentType(value);
}
else if (header.equals(DefaultHeaders.HOST_NAME)) {
    // get host name
}
else if (header.equals(DefaultHeaders.CONNECTION_NAME)) {
    if (header.valueEquals(DefaultHeaders.CONNECTION_CLOSE_VALUE)) {
        keepAlive = false;
        response.setHeader("Connection", "close");
    }
}
else if (header.equals(DefaultHeaders.EXPECT_NAME)) {
    if (header.valueEquals(DefaultHeaders.EXPECT_100_VALUE)) {
        sendAck = true;
    }
    else {
        throw new ServletException(sm.getString
            ("httpProcessor.parseHeaders.unknownExpectation"));
    }
}
else if (header.equals(DefaultHeaders.TRANSFER_ENCODING_NAME)) {
    //request.setTransferEncoding(header);
}

request.nextHeader();

```

## 4.8 简单的 Container 应用程序

本章应用程序的目的是说明如何使用默认连接器。该应用程序包括两个类: `ex04.pyrmont.core.SimpleContainer` 类和 `ex04.pyrmont.startup.Bootstrap` 类。`SimpleContainer` 类实现 `org.apache`.



catalina.Container 接口，这样它就可以与默认连接器进行关联。Bootstrap 类用于启动该应用程序。相比于第 3 章的应用程序，这里已经移除了连接器模块，以及对 ServletProcessor 类和 StaticResourceProcessor 类的使用，所以，你现在不能请求静态页面了。

代码清单 4-3 给出了 SimpleContainer 类的实现。

代码清单 4-3 SimpleContainer 类的实现

```
package ex04.pyrmont.core;

import java.beans.PropertyChangeListener;
import java.net.URL;
import java.net.URLClassLoader;
import java.net.URLStreamHandler;
import java.io.File;
import java.io.IOException;
import javax.naming.directory.DirContext;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.catalina.Cluster;
import org.apache.catalina.Container;
import org.apache.catalina.ContainerListener;
import org.apache.catalina.Loader;
import org.apache.catalina.Logger;
import org.apache.catalina.Manager;
import org.apache.catalina.Mapper;
import org.apache.catalina.Realm;
import org.apache.catalina.Request;
import org.apache.catalina.Response;

public class SimpleContainer implements Container {

    public static final String WEB_ROOT =
        System.getProperty("user.dir") + File.separator + "webroot";

    public SimpleContainer() { }
    public String getInfo() {
        return null;
    }
    public Loader getLoader() {
        return null;
    }
    public void setLoader(Loader loader) { }
    public Logger getLogger() {
        return null;
    }
    public void setLogger(Logger logger) { }
    public Manager getManager() {
        return null;
    }
    public void setManager(Manager manager) { }
    public Cluster getCluster() {
        return null;
    }
    public void setCluster(Cluster cluster) { }
```

```

public String getName() {
    return null;
}
public void setName(String name) { }
public Container getParent() {
    return null;
}
public void setParent(Container container) { }
public ClassLoader getParentClassLoader() {
    return null;
}
public void setParentClassLoader(ClassLoader parent) { }
public Realm getRealm() {
    return null;
}
public void setRealm(Realm realm) { }
public DirContext getResources() {
    return null;
}
public void setResources(DirContext resources) { }
public void addChild(Container child) { }
public void addContainerListener(ContainerListener listener) { }
public void addMapper(Mapper mapper) { }
public void addPropertyChangeListener(
    PropertyChangeListener listener) { }
public Container findChild(String name) {
    return null;
}
public Container[] findChildren() {
    return null;
}
public ContainerListener[] findContainerListeners() {
    return null;
}
public Mapper findMapper(String protocol) {
    return null;
}
public Mapper[] findMappers() {
    return null;
}
public void invoke(Request request, Response response)
    throws IOException, ServletException {

    String servletName = (HttpServletRequest)
request).getRequestURI();
    servletName = servletName.substring(servletName.lastIndexOf("/") +
1);
    URLClassLoader loader = null;
    try {
        URL[] urls = new URL[1];
        URLStreamHandler streamHandler = null;
        File classPath = new File(WEB_ROOT);
        String repository = (new URL("file", null,
classPath.getCanonicalPath() + File.separator)).toString();
        urls[0] = new URL(null, repository, streamHandler);
        loader = new URLClassLoader(urls);
    }
    catch (IOException e) {
        System.out.println(e.toString());
    }
    Class myClass = null;
    try {

```

```

        myClass = loader.loadClass(servletName);
    }
    catch (ClassNotFoundException e) {
        System.out.println(e.toString());
    }

    Servlet servlet = null;

    try {
        servlet = (Servlet) myClass.newInstance();
        servlet.service((HttpServletRequest) request,
            (HttpServletResponse) response);
    }
    catch (Exception e) {
        System.out.println(e.toString());
    }
    catch (Throwable e) {
        System.out.println(e.toString());
    }
}

```

```

public Container map(Request request, boolean update) {
    return null;
}
public void removeChild(Container child) { }
public void removeContainerListener(ContainerListener listener) { }
public void removeMapper(Mapper mapper) { }
public void removePropertyChangeListener(
    PropertyChangeListener listener) { }
}

```

这里仅仅给出了 SimpleContainer 类中 invoke() 方法的实现，因为默认连接器会调用该方法。invoke() 方法会创建一个类载入器，载入相关 servlet 类，并调用该 servlet 类的 service() 方法。该方法与第 3 章中的 ServletProcessor 类的 process() 方法类似。

代码清单 4-4 给出了 Bootstrap 类的实现。

代码清单 4-4 Bootstrap 类的实现代码

```

package ex04.pyrmont.startup;
import ex04.pyrmont.core.SimpleContainer;
import org.apache.catalina.connector.http.HttpConnector;

```

```

public final class Bootstrap {
    public static void main(String[] args) {
        HttpConnector connector = new HttpConnector();
        SimpleContainer container = new SimpleContainer();
        connector.setContainer(container);
        try {
            connector.initialize();
            connector.start();

            // make the application wait until we press any key.
            System.in.read();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```



Bootstrap 类的 `main()` 方法分别创建 `org.apache.catalina.connector.http.HttpConnector` 类和 `SimpleContainer` 类的一个实例，然后调用连接器的 `setContainer()` 方法将连接器和 `Servlet` 容器相关联。接下来，它调用连接器的 `initialize()` 和 `start()` 方法，这样连接器就可以准备处理 8080 端口上的 HTTP 请求了。

通过按控制台上的一个键可以终止应用程序。

## 运行应用程序

要在 Windows 平台下运行该应用程序，需要在工作目录下执行如下命令：

```
java -classpath ./lib/servlet.jar;./ ex04.pyrmont.startup.Bootstrap
```

若要在 Linux 平台下运行，需要使用冒号来代替库文件之间的分号：

```
java -classpath ./lib/servlet.jar:./ ex04.pyrmont.startup.Bootstrap
```

你可以像第 3 章中一样，调用 `PrimitiveServlet` 和 `ModernServlet` 资源。注意，现在你请求 `index.html` 文件时，无法获取到输出。因为应用程序中已经将处理静态资源的处理器移除了。

## 4.9 小结

本章展示了如何创建一个可以在 Catalina 中工作的 Tomcat 连接器，剖析了 Tomcat 4 的默认连接器的代码，并构建了一个小型应用程序来使用默认连接器。后面章节中的应用程序都会使用默认连接器。

## 第⑤章

# servlet 容器

servlet 容器是用来处理请求 servlet 资源，并为 Web 客户端填充 response 对象的模块。servlet 容器是 `org.apache.catalina.Container` 接口的实例。在 Tomcat 中，共有 4 种类型的容器，分别是：Engine、Host、Context 和 Wrapper。本章将对 Context 和 Wrapper 两种 servlet 容器进行讲解，Engine 和 Host 两类将会在第 13 章进行讲解。本章首先会介绍 Container 接口，讨论 servlet 容器中的管道机制。然后，介绍 Context 接口和 Wrapper 接口。本章末尾的两个应用程序分别展示了 Wrapper 实例和 Context 实例的使用方法。

### 5.1 Container 接口

Tomcat 中的 servlet 容器必须要实现 `org.apache.catalina.Container` 接口。正如在第 4 章中讲到的那样，需要将 servlet 容器的实例作为参数传入到连接器的 `setContainer()` 方法中，这样连接器才能调用 servlet 容器的 `invoke()` 方法。回忆一下，第 4 章的应用程序里面 Bootstrap 类中的如下代码：

```
HttpConnector connector = new HttpConnector();
SimpleContainer container = new SimpleContainer();
connector.setContainer(container);
```

对于 Catalina 中的 servlet 容器，首先需要注意的是，共有 4 种类型的容器，分别对应不同的概念层次。

- Engine：表示整个 Catalina servlet 引擎；
- Host：表示包含有一个或多个 Context 容器的虚拟主机；
- Context：表示一个 Web 应用程序。一个 Context 可以有多个 Wrapper；
- Wrapper：表示一个独立的 servlet。

上述的每个概念层级都由 `org.apache.catalina` 包内的一个接口表示，这些接口分别是 Engine、Host、Context 和 Wrapper，它们都继承自 Container 接口。这 4 个接口的标准实现分别是 StandardEngine 类、StandardHost 类、StandardContext 类和 StandardWrapper 类，它们都在 `org.apache.catalina.core` 包内。

图 5-1 展示了 Container 接口及其子接口和实现类的 UML 类图。注意，所有的接口都是 `org.apache.catalina` 包的一部分，所有的类都位于 `org.apache.catalina.core` 包下。

**注意** 所有的实现类都继承自抽象类 ContainerBase。

部署功能性的 Catalina 并不是必须将所有 4 种容器都包括在内。例如，本章中第 1 个应用

程序的 servlet 容器模块仅仅使用了一个 Wrapper 实例。第 2 个应用程序中的 servlet 容器模块使用了一个 Context 实例和一个 Wrapper 实例。本章所附的应用程序既不需要 Host 实例也不需要 Wrapper 实例。

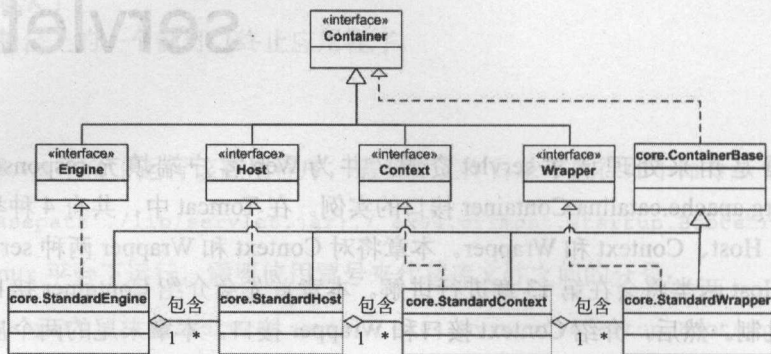


图 5-1 Container 接口及其相关类的 UML 类图

一个容器可以有 0 个或多个低层级的子容器。例如，一般情况下，一个 Context 实例会有一个或多个 Wrapper 实例，一个 Host 实例中会有 0 个或多个 Context 实例。但是，Wrapper 类型处于层级结构的最底层，因此，它无法再包含子容器了。可以通过调用 Container 接口的 addChild() 方法向某容器中添加子容器，该方法的签名如下：

```
public void addChild(Container child);
```

调用 Container 接口的 removeChild() 方法删除某容器中的一个子容器，remove() 方法的签名如下：

```
public void removeChild(Container child);
```

此外，Container 接口也提供了 findChild() 方法和 findChildren() 方法来查找某个子容器和所有子容器的某个集合。这两个方法的签名如下：

```
public Container findChild(String name);
public Container[] findChildren();
```

容器可以包含一些支持的组件，如载入器、记录器、管理器、领域和资源等，这些组件将在后续的章节中进行讨论。值得注意的是，Container 接口提供了 getter 和 setter 方法将这些组件与容器相关联。这些方法包括 getLoader() 和 setLoader(), getLogger() 和 setLogger(), getManager() 和 setManager(), getRealm() 和 setRealm(), 以及 getResources() 和 setResources()。

更有趣的是，Container 接口的设计满足以下条件：在部署应用时，Tomcat 管理员可以通过编辑配置文件 (server.xml) 来决定使用哪种容器。这是通过引入容器中的管道 (pipeline) 和阀 (valve) 的集合实现的，我们将在 5.2 节中对它们进行讨论。

**注意** Tomcat 4 中的 Container 接口与 Tomcat 5 中的有些许不同。例如，在 Tomcat 4 中 Container 接口有一个 map() 方法，而在 Tomcat 5 中 Container 接口中则没有。



## 5.2 管道任务

该节旨在说明当连接器调用了 servlet 容器的 `invoke()` 方法后会发生什么事。然后，在对应小节中讨论 `org.apache.catalina` 包中的 4 个相关接口，`Pipeline`、`Valve`、`ValveContext` 和 `Contained`。

管道包含该 servlet 容器将要调用的任务。一个阀表示一个具体的执行任务。在 servlet 容器的管道中，有一个基础阀，但是，可以添加任意数量的阀。阀的数量指的是额外添加的阀数量，即，不包括基础阀。有意思的是，可以通过编辑 Tomcat 的配置文件（`server.xml`）来动态地添加阀。图 5-2 显示了一条管道及其阀。

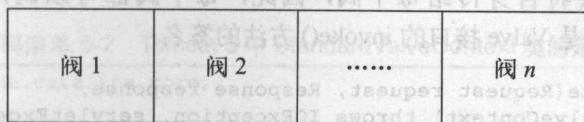


图 5-2 管道与阀

如果你对 servlet 编程中的过滤器有所了解的话，那么应该不难想象管道和阀的工作机制。管道就像过滤器链一样，而阀则好似是过滤器。阀与过滤器类似，可以处理传递给它的 `request` 对象和 `response` 对象。当一个阀执行完成后，会调用下一个阀继续执行。基础阀总是最后一个执行的。

一个 servlet 容器可以有一条管道。当调用了容器的 `invoke()` 方法后，容器会将处理工作交由管道完成，而管道会调用其中的第 1 个阀开始处理。当第 1 个阀处理完后，它会调用后续的阀继续执行任务，直到管道中所有的阀都处理完成。下面是在管道的 `invoke()` 方法中执行的伪代码：

```
// invoke each valve added to the pipeline
for (int n=0; n<valves.length; n++) {
    valve[n].invoke( ... );
}
// then, invoke the basic valve
basicValve.invoke( ... );
```

但是，Tomcat 的设计者选择了另一种实现方法，通过引入接口 `org.apache.catalina.ValveContext` 来实现阀的遍历执行。下面是它的工作原理。

当连接器调用容器的 `invoke()` 方法后，容器中要执行的任务并没有硬编码写在 `invoke()` 方法中。相反，容器会调用其管道的 `invoke()` 方法。`Pipeline` 接口的 `invoke()` 方法的签名与 `Container` 接口的 `invoke()` 方法完全相同，如下所示：

```
public void invoke(Request request, Response response)
    throws IOException, ServletException;
```

下面是 `Container` 接口在 `org.apache.catalina.core.ContainerBase` 类中的 `invoke()` 方法的实现：

```
public void invoke(Request request, Response response)
    throws IOException, ServletException {
    pipeline.invoke(request, response);
}
```

其中 `pipeline` 是该容器中的 `Pipeline` 接口的实例。

现在,管道必须保证添加到其中的所有阀及其基础阀都被调用一次,这是通过创建一个 ValveContext 接口实例来实现的。ValveContext 是作为管道的一个内部类实现的,因此,ValveContext 接口就可以访问管道的所有成员。ValveContext 接口中最重要的是 invokeNext():

```
public void invokeNext(Request request, Response response)
    throws IOException, ServletException;
```

在创建了 ValveContext 实例后,管道会调用 ValveContext 实例的 invokeNext() 方法。ValveContext 实例会首先调用管道中的第 1 个阀,第 1 个阀执行完后,会调用后面的阀继续执行。ValveContext 实例会将自身传给每个阀,因此,每个阀都可以调用 ValveContext 实例的 invokeNext() 方法。下面是 Valve 接口的 invoke() 方法的签名:

```
public void invoke(Request request, Response response,
    ValveContext ValveContext) throws IOException, ServletException
```

Valve 接口的 invoke() 方法的实现类似如下代码:

```
public void invoke(Request request, Response response,
    ValveContext valveContext) throws IOException, ServletException {
    // Pass the request and response on to the next valve in our pipeline
    valveContext.invokeNext(request, response);
    // now perform what this valve is supposed to do
    ...
}
```

org.apache.catalina.core.StandardPipeline 类是所有 servlet 容器的中 Pipeline 接口的实现。在 Tomcat 4 中,该类有一个实现了 ValveContext 接口的内部类,名为 StandardPipelineValveContext。代码清单 5-1 给出了 StandardPipelineValveContext 类的定义。

代码清单 5-1 Tomcat 4 中 StandardPipelineValveContext 类的定义

```
protected class StandardPipelineValveContext implements ValveContext {
    protected int stage = 0;
    public String getInfo() {
        return info;
    }
    public void invokeNext(Request request, Response response)
        throws IOException, ServletException {

        int subscript = stage;
        stage = stage + 1;
        // Invoke the requested Valve for the current request thread
        if (subscript < valves.length) {
            valves[subscript].invoke(request, response, this);
        }
        else if ((subscript == valves.length) && (basic != null)) {
            basic.invoke(request, response, this);
        }
        else {
            throw new ServletException
                (sm.getString("standardPipeline.noValve"));
        }
    }
}
```

invokeNext() 方法使用变量 subscript 和 stage 标明当前正在调用的阀。当第一次调用管道的 invoke() 方法时, subscript 的值为 0, stage 的值为 1, 因此, 第 1 个阀 (数组索引为 0) 会被调用。管道中的第 1 个阀接收 ValveContext 实例, 并调用其 invokeNext() 方法。这时, subscript 的值变为 1, 这样就会调用第 2 个阀, 此后以此类推。

当从最后一个阀调用 invokeNext() 方法时, subscript 的值等于阀的数量, 于是, 基础阀被调用。

Tomcat 5 从 StandardPipeline 类中移除了 StandardPipelineValveContext 类, 却使用 org.apache.catalina.core.StandardValveContext 类来调用阀, StandardValveContext 类的定义在代码清单 5-2 中给出。

代码清单 5-2 Tomcat 5 中 StandardValveContext 类的定义

```
package org.apache.catalina.core;

import java.io.IOException;
import javax.servlet.ServletException;
import org.apache.catalina.Request;
import org.apache.catalina.Response;
import org.apache.catalina.Valve;
import org.apache.catalina.ValveContext;
import org.apache.catalina.util.StringManager;

public final class StandardValveContext implements ValveContext {
    protected static StringManager sm =
        StringManager.getManager(Constants.Package);
    protected String info =
        "org.apache.catalina.core.StandardValveContext/1.0";
    protected int stage = 0;
    protected Valve basic = null;
    protected Valve valves[] = null;
    public String getInfo() {
        return info;
    }

    public final void invokeNext(Request request, Response response)
        throws IOException, ServletException {
        int subscript = stage;
        stage = stage + 1;
        // Invoke the requested Valve for the current request thread
        if (subscript < valves.length) {
            valves[subscript].invoke(request, response, this);
        }
        else if ((subscript == valves.length) && (basic != null)) {
            basic.invoke(request, response, this);
        }
        else {
            throw new ServletException
                (sm.getString("standardPipeline.noValve"));
        }
    }

    void set(Valve basic, Valve valves[]) {
        stage = 0;
        this.basic = basic;
        this.valves = valves;
    }
}
```



你能看出 Tomcat 4 中 *StandardPipelineValveContext* 类和 Tomcat 5 中的 *StandardValveContext* 类的相似点吗?

现在,我们要详细介绍几个接口,包括 Pipeline、Valve 和 ValveContext。此外,还会讨论一个阀类通常都会实现的接口 *org.apache.catalina.Contained*。

### 5.2.1 Pipeline 接口

对于 Pipeline 接口,首先要提到的一个方法是 *invoke()* 方法, *servlet* 容器调用 *invoke()* 方法来开始调用管道中的阀和基础阀。通过调用 Pipeline 接口的 *addValve()* 方法,可以向管道中添加新的阀,同样,也可以调用 *removeValve()* 方法从管道中删除某个阀。最后,调用 *setBasic()* 方法将基础阀设置到管道中,调用其 *getBasic()* 方法则可以获取基础阀。基础阀是最后调用的阀,负责处理 request 对象及其对应的 response 对象。代码清单 5-3 给出了 Pipeline 接口的定义。

代码清单 5-3 Pipeline 接口

```
package org.apache.catalina;
import java.io.IOException;
import javax.servlet.ServletException;

public interface Pipeline {
    public Valve getBasic();
    public void setBasic(Valve valve);
    public void addValve(Valve valve);
    public Valve[] getValves();
    public void invoke(Request request, Response response)
        throws IOException, ServletException;
    public void removeValve(Valve valve);
}
```

### 5.2.2 Valve 接口

阀是 Valve 接口的实例,用来处理接收到的请求。该接口有两个方法, *invoke()* 方法和 *getInfo()* 方法。*invoke()* 方法已经在前面讨论过了, *getInfo()* 方法返回阀的实现信息。代码清单 5-4 给出了 Valve 接口的定义。

代码清单 5-4 Valve 接口

```
package org.apache.catalina;
import java.io.IOException;
import javax.servlet.ServletException;

public interface Valve {
    public String getInfo();
    public void invoke(Request request, Response response,
        ValveContext context) throws IOException, ServletException;
}
```

### 5.2.3 ValveContext 接口

该接口有两个方法, *invokeNext()* 和 *getInfo()*。*invokeNext()* 方法已经在前面讨论过了,

getInfo() 方法会返回 ValveContext 的实现信息。代码清单 5-5 给出了 ValveContext 接口的定义。

代码清单 5-5 ValveContext 接口

```
package org.apache.catalina;
import java.io.IOException;
import javax.servlet.ServletException;

public interface ValveContext {
    public String getInfo();
    public void invokeNext(Request request, Response response)
        throws IOException, ServletException;
}
```

### 5.2.4 Contained 接口

阀可以选择是否实现 org.apache.catalina.Contained 接口，该接口的实现类可以通过接口中的方法至多与一个 servlet 容器相关联。代码清单 5-6 给出了 Contained 接口的定义。

代码清单 5-6 Contained 接口

```
package org.apache.catalina;
public interface Contained {
    public Container getContainer();
    public void setContainer(Container container);
}
```

## 5.3 Wrapper 接口

Wrapper 级的 servlet 容器是一个 org.apache.catalina.Wrapper 接口的实例，表示一个独立的 servlet 定义。Wrapper 接口继承自 Container 接口，又添加了一些额外的方法。Wrapper 接口的实现类要负责管理其基础 servlet 类的 servlet 生命周期，即，调用 servlet 的 init()、service()、destroy() 等方法。由于 Wrapper 已经是最低级的 servlet 容器了，因此不能再向其中添加子容器。若是 Wrapper 的 addChild() 方法被调用，则抛出 IllegalArgumentException 异常。

Wrapper 接口中比较重要的方法是 load() 和 allocate() 方法。allocate() 方法会分配一个已经初始化的 servlet 实例，而且，allocate() 方法还要考虑下该 servlet 类是否实现了 javax.servlet.SingleThreadModel 接口（将在第 11 章讨论）。load() 方法载入并初始化 servlet 类。这两个方法的签名如下：

```
public javax.servlet.Servlet allocate() throws
    javax.servlet.ServletException;
public void load() throws javax.servlet.ServletException;
```

Wrapper 接口中的其他方法将在第 11 章讨论 org.apache.catalina.core.StandardWrapper 类的时候介绍。

## 5.4 Context 接口

Context 接口的实例表示一个 Web 应用程序，一个 Context 实例可以有一个或多个 Wrapper 实例作为其子容器。

Context 接口中比较重要的方法是 addWrapper() 方法和 createWrapper() 方法。Context 接口的更多细节将在第 12 章中讨论。

## 5.5 Wrapper 应用程序

本节的应用程序展示如何编写一个最小的 servlet 容器。应用程序的核心类是 ex05.pyrmont.core.SimpleWrapper，该类实现了 Wrapper 接口。SimpleWrapper 类包含一个 Pipeline 实例 (ex05.pyrmont.core.SimplePipeline 类的实例)，并使用一个 Loader 实例 (ex05.pyrmont.core.SimpleLoader 类的实例) 来载入 servlet 类。Pipeline 实例包含了一个基础阀 (ex05.pyrmont.core.SimpleWrapperValve 类) 和两个额外的阀 (ex05.pyrmont.core.ClientPLoggerValve 类和 ex05.pyrmont.core.HeaderLoggerValve 类)。图 5-3 给出了该应用程序的类图。

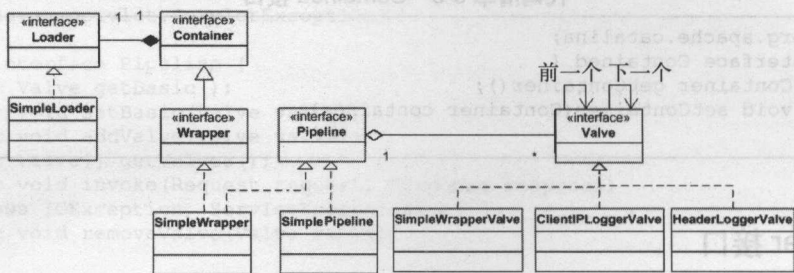


图 5-3 Wrapper 应用程序的 UML 类图

**注意** servlet 容器使用 Tomcat 4 中的默认连接器。

Wrapper 实例将之前章节中使用过的 ModernServlet 类进行包装。在应用程序中，servlet 容器中只会有一个 Wrapper 实例。其中使用到的类并没有完全开发，只实现了一些必须用到的方法。下面将对这些类的细节进行介绍。

### 5.5.1 ex05.pyrmont.core.SimpleLoader 类

在 servlet 容器中载入相关 servlet 类的工作由 Loader 接口的实例完成。在本应用程序中，SimpleLoader 负责完成类的载入工作。它知道 servlet 类的位置，通过调用其 getClassLoader() 方法可以返回一个 java.lang.ClassLoader 实例，可用来搜索 servlet 类的位置。SimpleLoader 类声明了 3 个变量。第 1 个是 WEB\_ROOT，指明了查找 servlet 类所在的目录：

```
public static final String WEB_ROOT =
    System.getProperty("user.dir") + File.separator + "webroot";
```

其他两个变量分别是 ClassLoader 和 Container 类型的对象引用：



```
ClassLoader classLoader = null;
Container container = null;
```

SimpleLoader 类的构造函数会初始化类加载器，供 SimpleWrapper 实例使用：

```
public SimpleLoader() {
    try {
        URL[] urls = new URL[1];
        URLStreamHandler streamHandler = null;
        File classPath = new File(WEB_ROOT);
        String repository = (new URL("file", null,
            classPath.getCanonicalPath() + File.separator)).toString();
        urls[0] = new URL(null, repository, streamHandler);
        classLoader = new URLClassLoader(urls);
    }
    catch (IOException e) {
        System.out.println(e.toString());
    }
}
```

构造函数中的代码会初始化一个类加载器，在前面的章节中已经解释过，这里不再重复。

其中的变量 container 指向了与该 servlet 容器相关联的类加载器。

**注意** 载入器的细节将在第 8 章中讨论。

### 5.5.2 ex05.pyrmont.core.SimplePipeline 类

SimplePipeline 类实现 org.apache.catalina.Pipeline 接口，该类中最重要的方法是 invoke() 方法，该方法包含了一个内部类 SimplePipelineValveContext，该类实现 org.apache.catalina.ValveContext 接口。ValveContext 接口已经在 5.2 节中介绍过了。

### 5.5.3 ex05.pyrmont.core.SimpleWrapper 类

该类实现 org.apache.catalina.Wrapper 接口，并提供 allocate() 和 load() 方法的实现。此外，该类还声明两个变量：

```
private Loader loader;
protected Container parent = null;
```

loader 变量指明了载入 servlet 类要使用的载入器。parent 变量指明该 Wrapper 实例的父容器，即，Wrapper 实例可以成为其他容器的子容器，例如，成为 Context 实例的子容器。

注意其中的 getLoader() 方法，代码清单 5-7 给出了 getLoader() 方法的实现。

代码清单 5-7 SimpleWrapper 类的 getLoader 方法的实现

```
public Loader getLoader() {
    if (loader != null)
        return (loader);
    if (parent != null)
        return (parent.getLoader());
    return (null);
}
```

该方法返回一个用于载入 servlet 类的载入器。若 Wrapper 实例已经关联了一个载入器，则直接将其返回；否则，它将返回父容器的载入器。若没有父容器，getLoader() 方法会返回 null。

SimpleWrapper 类有一个 Pipeline 实例，并为该 Pipeline 实例设置了基础阀。这些都是在 SimpleWrapper 类的构造函数中完成的，如代码清单 5-8 所示。

代码清单 5-8 SimpleWrapper 类的构造函数

---

```
public SimpleWrapper() {
    pipeline.setBasic(new SimpleWrapperValve());
}
```

---

这里，变量 pipeline 是 SimplePipeline 类的实例，是一个成员变量：

```
private SimplePipeline pipeline = new SimplePipeline(this);
```

#### 5.5.4 ex05.pyrmont.core.SimpleWrapperValve 类

SimpleWrapperValve 是一个基础阀，专门用于处理对 SimpleWrapper 类的请求。SimpleWrapperValve 类实现 org.apache.catalina.Valve 接口和 org.apache.catalina.Contained 接口。SimpleWrapperValve 类中最主要的方法是 invoke() 方法，其具体实现在代码清单 5-9 中给出。

代码清单 5-9 SimpleWrapperValve 类的 invoke() 方法的实现

---

```
public void invoke(Request request, Response response,
    ValveContext valveContext)
    throws IOException, ServletException {
    SimpleWrapper wrapper = (SimpleWrapper) getContainer();
    ServletRequest sreq = request.getRequest();
    ServletResponse sres = response.getResponse();
    Servlet servlet = null;
    HttpServletRequest hreq = null;
    if (sreq instanceof HttpServletRequest)
        hreq = (HttpServletRequest) sreq;
    HttpServletResponse hres = null;
    if (sres instanceof HttpServletResponse)
        hres = (HttpServletResponse) sres;
    // Allocate a servlet instance to process this request
    try {
        servlet = wrapper.allocate();
        if (hres != null && hreq != null) {
            servlet.service(hreq, hres);
        }
        else {
            servlet.service(sreq, sres);
        }
    }
    catch (ServletException e) {
    }
}
```

---

由于 SimpleWrapperValve 类是一个基础阀，因此它的 invoke() 方法不需要调用传递给它的 ValveContext 实例的 invokeNext() 方法。invoke() 方法会调用 SimpleWrapper 类的 allocate() 方法

来获取 Wrapper 实例表示的 servlet 的实例。然后，它调用该 servlet 实例的 service() 方法。注意，是 Wrapper 实例的基础阀调用了 servlet 实例的 service() 方法，而不是 Wrapper 实例本身调用的。

### 5.5.5 ex05.pyrmont.valves.ClientIPLoggerValve 类

ClientIPLoggerValve 类所表示的阀用来将客户端的 IP 地址输出到控制台上。代码清单 5-10 给出该类的定义。

代码清单 5-10 ClientIPLoggerValve 类的定义

```
package ex05.pyrmont.valves;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import org.apache.catalina.Request;
import org.apache.catalina.Response;
import org.apache.catalina.Valve;
import org.apache.catalina.ValveContext;
import org.apache.catalina.Contained;
import org.apache.catalina.Container;

public class ClientIPLoggerValve implements Valve, Contained {
    protected Container container;

    public void invoke(Request request, Response response,
        ValveContext valveContext) throws IOException, ServletException {
        // Pass this request on to the next valve in our pipeline
        valveContext.invokeNext(request, response);
        System.out.println("Client IP Logger Valve");
        ServletRequest sreq = request.getRequest();
        System.out.println(sreq.getRemoteAddr());
        System.out.println("-----");
    }

    public String getInfo() {
        return null;
    }

    public Container getContainer() {
        return container;
    }

    public void setContainer(Container container) {
        this.container = container;
    }
}
```

注意其 invoke() 方法，它先调用方法参数 valveContext 的 invokeNext() 方法来调用管道中的下一个阀。然后，它会把几行字符串（包括 getRemoteAddr() 方法的输出）输出到控制台上。

### 5.5.6 ex05.pyrmont.valves.HeaderLoggerValve 类

该类与 ClientIPLoggerValve 类类似。HeaderLoggerValve 类会把请求头信息输出到控制台。代码清单 5-11 给出了该类的定义。



代码清单 5-11 HeaderLoggerValve 类的定义

```

package ex05.pyrmont.valves;

import java.io.IOException;
import java.util.Enumeration;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import org.apache.catalina.Request;
import org.apache.catalina.Response;
import org.apache.catalina.Valve;
import org.apache.catalina.ValveContext;
import org.apache.catalina.Container;

public class HeaderLoggerValve implements Valve, Container {
    protected Container container;

    public void invoke(Request request, Response response,
        ValveContext valveContext) throws IOException, ServletException {

        // Pass this request on to the next valve in our pipeline
        valveContext.invokeNext(request, response);
        System.out.println("Header Logger Valve");
        ServletRequest sreq = request.getRequest();
        if (sreq instanceof HttpServletRequest) {
            HttpServletRequest hreq = (HttpServletRequest) sreq;
            Enumeration headerNames = hreq.getHeaderNames();
            while (headerNames.hasMoreElements()) {
                String headerName = headerNames.nextElement().toString();
                String headerValue = hreq.getHeader(headerName);
                System.out.println(headerName + ":" + headerValue);
            }
        }
        else
            System.out.println("Not an HTTP Request");

        System.out.println("-----");
    }

    public String getInfo() {
        return null;
    }

    public Container getContainer() {
        return container;
    }

    public void setContainer(Container container) {
        this.container = container;
    }
}

```

请再次注意 invoke() 方法。invoke() 方法会先调用参数 valveContext 的 invokeNext() 方法，以调用管道中的下一个阀，如果存在的话。然后它输出请求头信息。

### 5.5.7 ex05.pyrmont.startup.Bootstrap1

该类用于启动应用程序，Bootstrap1 类的定义在代码清单 5-12 中给出。

代码清单 5-12 Bootstrap1 类的定义

```

package ex05.pyrmont.startup;
import ex05.pyrmont.core.SimpleLoader;
import ex05.pyrmont.core.SimpleWrapper;
import ex05.pyrmont.valves.ClientIPLoggerValve;
import ex05.pyrmont.valves.HeaderLoggerValve;
import org.apache.catalina.Loader;
import org.apache.catalina.Pipeline;
import org.apache.catalina.Valve;
import org.apache.catalina.Wrapper;
import org.apache.catalina.connector.http.HttpConnector;

public final class Bootstrap1 {
    public static void main(String[] args) {
        HttpConnector connector = new HttpConnector();
        Wrapper wrapper = new SimpleWrapper();
        wrapper.setServletClass("ModernServlet");
        Loader loader = new SimpleLoader();
        Valve valve1 = new HeaderLoggerValve();
        Valve valve2 = new ClientIPLoggerValve();

        wrapper.setLoader(loader);
        ((Pipeline) wrapper).addValve(valve1);
        ((Pipeline) wrapper).addValve(valve2);

        connector.setContainer(wrapper);

        try {
            connector.initialize();
            connector.start();

            // make the application wait until we press a key.
            System.in.read();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

在创建了 `HttpConnector` 实例和 `SimpleWrapper` 实例后, `Bootstrap` 类的 `main()` 方法将字符串 “ModernServlet” 赋值给 `SimpleWrapper` 类的 `setServletClass()` 方法, 告诉 `Wrapper` 实例要载入的 servlet 类的名称:

```
wrapper.setServletClass("Modernservlet");
```

然后, `main()` 方法会创建一个载入器和两个阀, 并将载入器设置到 `Wrapper` 实例中:

```

Loader loader = new SimpleLoader();
Valve valve1 = new HeaderLoggerValve();
Valve valve2 = new ClientIPLoggerValve();
wrapper.setLoader(loader);

```

再将创建的两个阀添加到 `Wrapper` 实例的管道中:

```

((Pipeline) wrapper).addValve(valve1);
((Pipeline) wrapper).addValve(valve2);

```

最后，将 Wrapper 实例设置为连接器的 servlet 容器，并初始化并启动连接器：

```
connector.setContainer(wrapper);
```

```
try {
    connector.initialize();
    connector.start();
}
```

下面的代码使用户可以通过在控制台中按 Enter 键来关闭应用程序：

```
// make the application wait until we press Enter.
System.in.read();
```

### 5.5.8 运行应用程序

要在 Windows 平台下运行该应用程序，需要在工作目录下执行如下命令：

```
java -classpath ./lib/servlet.jar;./ ex05.pyrmont.startup.Bootstrap1
```

要在 Linux 平台下运行，需要使用冒号来代替库文件之间的分号：

```
java -classpath ./lib/servlet.jar:./ ex05.pyrmont.startup.Bootstrap1
```

可以使用下面的 URL 来调用 servlet：

```
http://localhost:8080
```

浏览器会显示 ModernServlet 实例的响应信息。控制台上的响应信息与下面类似：

```
ModernServlet -- init
Client IP Logger Valve
127.0.0.1
```

```
-----
Header Logger Valve
```

```
accept:image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/vnd.ms-excel, application/msword, application/vnd.ms-
powerpoint, */*
```

```
accept-language:en-us
```

```
accept-encoding:gzip, deflate
```

```
user-agent:Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR
```

```
1.1.4322)
```

```
host:localhost:8080
```

```
connection:Keep-Alive
-----
```

## 5.6 Context 应用程序

在本章的第 1 个应用程序中，你已经学会了如何部署一个仅包含一个 Wrapper 实例的简单 Web 应用程序。这个应用程序中只有一个 servlet 实例。虽然有些应用程序只需要一个 servlet，但大部分 Web 应用程序是需要多个 servlet 合作的。在这类应用程序中，需要的 servlet 容器是 Context，不是 Wrapper。

本章的第 2 个应用程序展示了如何使用一个包含了两个 Wrapper 实例的 Context 实例来构建 Web 应用程序，这两个 Wrapper 实例包装了两个 servlet 类。当应用程序中有多个 Wrapper 实例时，需要使用一个映射器。映射器是组件，帮助 servlet 容器——在本应用程序中是 Context 实例——选择一个子容器来处理某个指定的请求。



注意 映射器只存在于 Tomcat 4 中，Tomcat 5 使用了另一种方法来查找子容器。

本应用程序中的映射器是 `ex05.pyrmont.core.SimpleContextMapper` 类的一个实例，该类实现了 Tomcat 4 中的 `org.apache.catalina.Mapper` 接口。Servlet 容器可以使用多个映射器来支持不同的协议。在这里，一个映射器支持一个请求协议。例如，Servlet 容器中可以使用一个映射器对 HTTP 协议的请求进行映射，使用另一个对 HTTPS 协议的请求进行映射。代码清单 5-13 给出了 Tomcat 4 中 `Mapper` 接口的定义。

代码清单 5-13 Tomcat 4 中 Mapper 接口的定义

```
package org.apache.catalina;
public interface Mapper {
    public Container getContainer();
    public void setContainer(Container container);
    public String getProtocol();
    public void setProtocol(String protocol);
    public Container map(Request request, boolean update);
}
```

`Mapper` 接口的 `getContainer()` 方法返回与该映射器相关联的 Servlet 容器的实例，而 `setContainer()` 方法则用来将某个 Servlet 容器与该映射器相关联。`getProtocol()` 方法返回该映射器负责处理的协议，`setProtocol()` 方法可以指定该映射器负责处理哪种协议。`map()` 方法返回要处理某个特定请求的子容器的实例。

图 5-4 展示了本应用程序的类图。

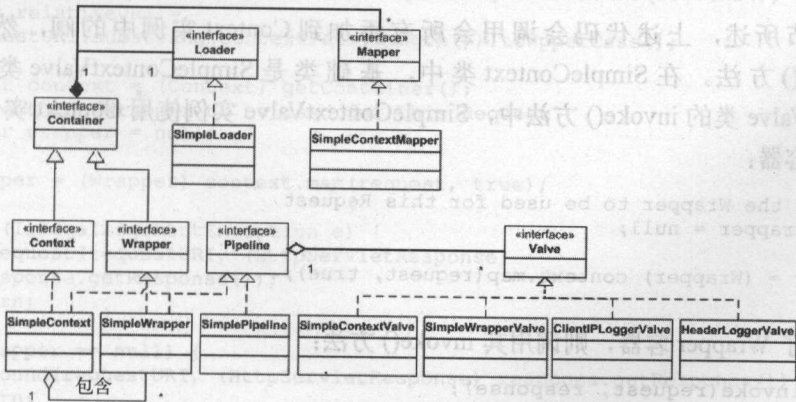


图 5-4 Context 应用程序相关类的 UML 类图

Context 容器是一个 `SimpleContext` 类的实例，`SimpleContext` 类使用 `SimpleContextMapper` 类的实例作为其映射器，将 `SimpleContextValve` 的实例作为其基础阀。Context 容器中额外添加了两个阀：`ClientIPLoggerValve` 和 `HeaderLoggerValve`，并包含两个 `Wrapper` 实例作为其子容器，二者都是 `SimpleWrapper` 实例。这两个 `Wrapper` 实例使用 `SimpleWrapperValve` 实例作为其基础阀，不再添加其他的阀。

Context 应用程序使用了与第 1 个应用程序相同的载入器和两个阀。但是，载入器与阀都

是与 Context 容器（而非 Wrapper）相关联的。这样，两个 Wrapper 实例就都可以使用载入器。Context 实例作为 servlet 容器被设置到连接器中。这样，当连接器接收到 HTTP 请求后，它就可以调用 Context 容器的 invoke() 方法了。回忆一下前面章节的内容，就应该可以理解剩余的内容了：

- 1) 容器包含一条管道，容器的 invoke() 方法会调用管道的 invoke() 方法；
- 2) 管道的 invoke() 方法会调用所有添加到其容器中的阀，然后再调用其基础阀的 invoke() 方法；
- 3) 在 Wrapper 实例中，基础阀负责载入相关联的 servlet 类，并对请求进行响应；
- 4) 在包含子容器的 Context 实例中，基础阀使用映射器来查找一个子容器，该子容器负责处理接收到的请求。若找到了相应的子容器，则调用其 invoke() 方法，转到步骤 1) 继续执行。

下面分析一下应用程序中上述顺序的实现：

SimpleContext 类的 invoke() 方法调用管道的 invoke() 方法：

```
public void invoke(Request request, Response response)
    throws IOException, ServletException {
    pipeline.invoke(request, response);
}
```

管道是 SimplePipeline 类的实例，它的 invoke() 方法如下：

```
public void invoke(Request request, Response response)
    throws IOException, ServletException {
    // Invoke the first Valve in this pipeline for this request
    (new SimplePipelineValveContext()).invokeNext(request, response);
}
```

正如 5.2 节所述，上述代码会调用会所有添加到 Context 实例中的阀，然后再调用基础阀的 invoke() 方法。在 SimpleContext 类中，基础类是 SimpleContextValve 类的实例。在 SimpleContextValve 类的 invoke() 方法中，SimpleContextValve 实例使用 Context 实例的映射器来查找 Wrapper 容器：

```
// Select the Wrapper to be used for this Request
Wrapper wrapper = null;
try {
    wrapper = (Wrapper) context.map(request, true);
}
```

如果找到了 Wrapper 容器，则调用其 invoke() 方法：

```
wrapper.invoke(request, response);
```

本应用程序的 Wrapper 实例是 SimpleWrapper 类的对象。下面的代码是 SimpleWrapper 类的 invoke() 方法，与 SimpleContext 类的 invoke() 方法完全相同：

```
public void invoke(Request request, Response response)
    throws IOException, ServletException {
    pipeline.invoke(request, response);
}
```

SimplePipeline 类的 invoke() 方法已经在上面列出来了，该管道是 SimplePipeline 的一个实例。应用程序中的 Wrapper 实例只有基础阀，也就是 SimpleWrapperValve 类的实例。正如 5.5 节

所述, Wrapper 实例的管道会调用 SimpleWrapperValve 类的 invoke() 方法, 它会分配 servlet 实例, 并调用其 service() 方法。

注意, Wrapper 实例中并没有与载入器相关联, 但是 Context 实例关联了类载入器。因此, SimpleWrapper 类的 getLoader() 方法会返回父容器的载入器。

下面 4 节会分别对 SimpleContext、SimpleContextValve、SimpleContextMapper 和 Bootstrap2 这 4 个类进行讨论。

### 5.6.1 ex05.pyrmont.core.SimpleContextValve 类

SimpleContext 实例的基础阀是 SimpleContextValve 类的实例。SimpleContextValve 类最重要的方法是 invoke() 方法, 其具体实现在代码清单 5-14 中给出。

代码清单 5-14 SimpleContextValve 类的 invoke() 方法的实现

```
public void invoke(Request request, Response response,
    ValveContext valveContext)
    throws IOException, ServletException {
    // Validate the request and response object types
    if (!(request.getRequest() instanceof HttpServletRequest) ||
        !(response.getResponse() instanceof HttpServletResponse)) {
        return;
    }

    // Disallow any direct access to resources under WEB-INF or META-INF
    HttpServletRequest hreq = (HttpServletRequest) request.getRequest();
    String contextPath = hreq.getContextPath();
    String requestURI = ((HttpRequest) request).getDecodedRequestURI();
    String relativeURI =
        requestURI.substring(contextPath.length()).toUpperCase();

    Context context = (Context) getContainer();
    // Select the Wrapper to be used for this Request
    Wrapper wrapper = null;
    try {
        wrapper = (Wrapper) context.map(request, true);
    }
    catch (IllegalArgumentException e) {
        badRequest(requestURI, (HttpServletResponse)
            response.getResponse());
        return;
    }
    if (wrapper == null) {
        notFound(requestURI, (HttpServletResponse) response.getResponse());
        return;
    }
    // Ask this Wrapper to process this Request
    response.setContext(context);
    wrapper.invoke(request, response);
}
```

### 5.6.2 ex05.pyrmont.core.SimpleContextMapper 类

代码清单 5-15 中的 SimpleContextMapper 类实现 Tomcat 4 中的 org.apache.catalina.Mapper



接口，需要与 SimpleContext 的实例相关联。

代码清单 5-15 SimpleContext 类的定义

```
package ex05.pyrmont.core;

import javax.servlet.http.HttpServletRequest;
import org.apache.catalina.Container;
import org.apache.catalina.HttpRequest;
import org.apache.catalina.Mapper;
import org.apache.catalina.Request;
import org.apache.catalina.Wrapper;

public class SimpleContextMapper implements Mapper {

    private SimpleContext context = null;

    public Container getContainer() {
        return (context);
    }

    public void setContainer(Container container) {
        if (!(container instanceof SimpleContext))
            throw new IllegalArgumentException(
                "Illegal type of container");
        context = (SimpleContext) container;
    }

    public String getProtocol() {
        return null;
    }

    public void setProtocol(String protocol) { }

    public Container map(Request request, boolean update) {
        // Identify the context-relative URI to be mapped
        String contextPath =
            ((HttpServletRequest) request.getRequest()).getContextPath();
        String requestURI = ((HttpRequest) request).getDecodedRequestURI();
        String relativeURI = requestURI.substring(contextPath.length());
        // Apply the standard request URI mapping rules from
        // the specification
        Wrapper wrapper = null;
        String servletPath = relativeURI;
        String pathInfo = null;
        String name = context.findServletMapping(relativeURI);
        if (name != null)
            wrapper = (Wrapper) context.findChild(name);
        return (wrapper);
    }
}
```

如果向 setContainer() 方法中传入的对象不是 SimpleContext 类的实例，则它会抛出 IllegalArgumentException 异常。map() 方法会返回一个子容器（也就是 Wrapper 实例）来处理请求。map() 方法需要两个参数，一个 request 对象和一个布尔变量。在本应用程序中，忽略了第 2 个参数。map() 方法会从 request 对象中解析出请求的上下文路径，并调用 Context 实例的 findServletMapping() 方法来获取一个与该路径相关联的名称。如果找到了这个名称，则它调用 Context 实例的 findChild() 获取一个 Wrapper 实例。

### 5.6.3 ex05.pyrmont.core.SimpleContext 类

在本应用程序中，Context 容器是 SimpleContext 类的实例，是与连接器相关的主容器。但是对每个独立 servlet 的处理是由 Wrapper 实例完成的。本应用程序有两个 servlet 实例：PrimitiveServlet 和 ModernServlet 因此有两个 Wrapper 实例。每个 Wrapper 实例都有对应的名称，PrimitiveServlet 类对应的 Wrapper 实例的名称是 Primitive，ModernServlet 类对应的 Wrapper 实例的名称是 Modern。对每个请求来说，SimpleContext 实例是通过请求的 URL 模式与 Wrapper 实例名称之间的映射来决定调用哪个 Wrapper 实例的。本应用程序有两种 URL 模式可以用来调用两个 Wrapper 实例，其中“/Primitive”模式会映射到 Primitive，“/Modern”模式会映射到 Modern。当然，也可以将多个 URL 模式映射到一个 Wrapper 实例上。只须添加这些模式即可。

SimpleContext 类必须实现 Container 和 Context 接口，其中声明的很多方法在本应用程序中都留空了。但与映射相关的方法都给出了具体的实现代码。这些方法包括以下几个：

- addServletMapping(), 添加一个 URL 模式 /Wrapper 实例的名称对；通过给定的名称添加用于调用 Wrapper 实例的每种模式。
- findServletMapping(), 通过 URL 模式查找对应的 Wrapper 实例名称。该方法用来查找某个特殊 URL 模式对应的 Wrapper 实例。如果给定的 URL 模式并没有使用 addServletMapping() 方法注册，则 findServletMapping() 方法返回 null；
- addMapper(), 在 Context 容器中添加一个映射器。SimpleContext 类声明两个变量：mapper 和 mappers。mapper 表示程序使用的默认映射器，mappers 包含 SimpleContext 实例中所有可用的映射器。第一个被添加到 Context 容器中的映射器成为默认映射器；
- findMapper(), 找到正确的映射器，在 SimpleContext 类中，它返回默认映射器；
- map(), 返回负责处理当前请求的 Wrapper 实例。

### 5.6.4 ex05.pyrmont.startup.Bootstrap2

该类用来启动应用程序，与 Bootstrap1 类相似。Bootstrap2 类的定义在代码清单 5-16 中给出。

代码清单 5-16 Bootstrap2 类的定义

```
package ex05.pyrmont.startup;
import ex05.pyrmont.core.SimpleContext;
import ex05.pyrmont.core.SimpleContextMapper;
import ex05.pyrmont.core.SimpleLoader;
import ex05.pyrmont.core.SimpleWrapper;
import ex05.pyrmont.valves.ClientIPLoggerValve;
import ex05.pyrmont.valves.HeaderLoggerValve;
import org.apache.catalina.Connector;
import org.apache.catalina.Context;
import org.apache.catalina.Loader;
import org.apache.catalina.Mapper;
import org.apache.catalina.Pipeline;
import org.apache.catalina.Valve;
import org.apache.catalina.Wrapper;
import org.apache.catalina.connector.http.HttpConnector;

public final class Bootstrap2 {
    public static void main(String[] args) {
```

```

HttpConnector connector = new HttpConnector();
Wrapper wrapper1 = new SimpleWrapper();
wrapper1.setName("Primitive");
wrapper1.setServletClass("PrimitiveServlet");
Wrapper wrapper2 = new SimpleWrapper();
wrapper2.setName("Modern");
wrapper2.setServletClass("ModernServlet");

Context context = new SimpleContext();
context.addChild(wrapper1);
context.addChild(wrapper2);

Valve valve1 = new HeaderLoggerValve();
Valve valve2 = new ClientIPLoggerValve();

((Pipeline) context).addValve(valve1);
((Pipeline) context).addValve(valve2);

Mapper mapper = new SimpleContextMapper();
mapper.setProtocol("http");
context.addMapper(mapper);
Loader loader = new SimpleLoader();
context.setLoader(loader);
// context.addServletMapping(pattern, name);
context.addServletMapping("/Primitive", "Primitive");
context.addServletMapping("/Modern", "Modern");
connector.setContainer(context);
try {
    connector.initialize();
    connector.start();

    // make the application wait until we press a key.
    System.in.read();
}
catch (Exception e) {
    e.printStackTrace();
}
}
}

```

其中，main() 方法首先会实例化 Tomcat 的默认连接器，创建两个 Wrapper 实例，分别是 wrapper1 和 wrapper2，并分别指定名称为 Primitive 和 Modern。Modern 对应的 servlet 类是 ModernServlet，Primitive 对应的 servlet 类是 PrimitiveServlet。

```

HttpConnector connector = new HttpConnector();
Wrapper wrapper1 = new SimpleWrapper();
wrapper1.setName("Primitive");
wrapper1.setServletClass("PrimitiveServlet");
Wrapper wrapper2 = new SimpleWrapper();
wrapper2.setName("Modern");
wrapper2.setServletClass("ModernServlet");

```

然后，main() 方法创建一个 SimpleContext 实例，并将 wrapper1 和 wrapper2 作为子容器添加到 SimpleContext 实例中。此外，它还会实例化两个阀：ClientIPLoggerValve 和 HeaderLoggerValve，并将它们添加到 SimpleContext 实例中：



```
Context context = new SimpleContext();
context.addChild(wrapper1);
context.addChild(wrapper2);
```

```
Valve valve1 = new HeaderLoggerValve();
Valve valve2 = new ClientIPLoggerValve();

((Pipeline) context).addValve(valve1);
((Pipeline) context).addValve(valve2);
```

接下来，它会从 SimpleMapper 类创建一个映射器对象，将其添加到 SimpleContext 实例中。映射器负责查找 Context 实例中的子容器来处理 HTTP 请求：

```
Mapper mapper = new SimpleContextMapper();
mapper.setProtocol("http");
context.addMapper(mapper);
```

要载入 servlet 类，还需要一个载入器。这里会用到 SimpleLoader 类，就像第 1 个应用程序那样。但是，这里并没有将其添加到 Wrapper 实例中，而是把它添加到 Context 实例中。Wrapper 实例可以通过其 getLoader() 方法来获取载入器，因为 Wrapper 实例是 Context 实例的子容器：

```
Loader loader = new SimpleLoader();
context.setLoader(loader);
```

现在，需要添加 servlet 映射了。为两个 Wrapper 实例添加两种模式。

```
// context.addServletMapping(pattern, name);
context.addServletMapping("/Primitive", "Primitive");
context.addServletMapping("/Modern", "Modern");
```

最后，将 Context 容器与连接器相关联，并初始化连接器，调用其 start() 方法：

```
connector.setContainer(context);
try {
    connector.initialize();
    connector.start();
}
```

## 5.6.5 运行应用程序

要在 Windows 平台下运行该应用程序，需要在工作目录下执行如下命令：

```
java -classpath ./lib/servlet.jar;./ ex05.pyrmont.startup.Bootstrap2
```

要在 Linux 平台下运行，需要使用冒号来代替库文件之间的分号：

```
java -classpath ./lib/servlet.jar:./ ex05.pyrmont.startup.Bootstrap2
```

要调用 PrimitiveServlet，可以使用如下的 URL：

```
http://localhost:8080/Primitive
```

类似地，要调用 ModernServlet 时，可以使用如下的 URL：

```
http://localhost:8080/Modern
```

## 5.7 小结

Servlet 容器是整个 Tomcat 的第 2 个主要模块（第 1 个是连接器）。容器还包含有其他很多模块，如记录器、载入器和管理器等。共有 4 种类型的容器，分别是：Engine、Host、Context 和 Wrapper。而在实际部署中，4 种容器并不是都需要。本章中的两个应用程序分别展示了如何部署只使用 1 个 Wrapper 实例或 1 个 Context 实例带有几个 Wrapper 实例的应用。

```

// 创建 Context 实例
Context context = new SimpleContext();
context.addChild(wrapper);

// 创建 Wrapper 实例
Wrapper wrapper = new SimpleContextWrapper(context);
wrapper.setProtocol("http");
context.addChild(wrapper);

// 创建 Servlet 实例
Servlet servlet = new SimpleServlet();
context.addChild(servlet);

// 启动 Context
context.start();

// 启动 Wrapper
wrapper.start();

// 启动 Servlet
servlet.start();

// 启动连接器
connector.setContext(context);
try {
    connector.initialize();
    connector.start();
} catch (Exception e) {
    e.printStackTrace();
}

```

### 5.7.5 运行应用程序

其中，main() 命令会执行以下操作：首先，它会在类路径中查找名为 SimpleContext 的类，并实例化它。然后，它会在类路径中查找名为 SimpleServlet 的类，并实例化它。最后，它会在类路径中查找名为 SimpleContextWrapper 的类，并实例化它。此外，它还会实例化两个 Context 实例，并将它们添加到 SimpleContext 实例中。

## 第 ⑥ 章

# 生命周期

Catalina 包含有很多组件。当 Catalina 启动时，这些组件也会一起启动，同样，当 Catalina 关闭时，这些组件也会随之关闭。例如，当 servlet 容器关闭时，它必须调用所有已经载入到容器中的 servlet 类的 `destroy()` 方法，而 Session 管理器必须将 Session 对象保存到辅助存储器中。通过实现 `org.apache.catalina.Lifecycle` 接口，可以达到统一启动 / 关闭这些组件的效果。

实现了 `org.apache.catalina.Lifecycle` 接口的组件可以触发一个或多个下面的事件：`BEFORE_START_EVENT`、`START_EVENT`、`AFTER_START_EVENT`、`BEFORE_STOP_EVENT`、`STOP_EVENT`、`AFTER_STOP_EVENT`。当组件启动时，正常会触发前 3 个事件；而关闭组件时，会触发后 3 个事件。事件是 `org.apache.catalina.LifecycleEvent` 类的实例。当然，如果 Catalina 组件可以触发事件，那么需要编写相应的事件监听器对这些事件进行响应。事件监听器是 `org.apache.catalina.LifecycleListener` 接口的实例。

本章将会对 3 种类型进行讨论，分别是 `Lifecycle`、`LifecycleEvent` 和 `LifecycleListener`。此外，还会介绍一个工具类 `LifecycleSupport`。该类提供了一种简单的方法来触发某个组件的生命周期事件，并对事件监听器进行处理。本章中的应用程序会基于第 5 章的应用程序进行开发，使用一些实现了 `Lifecycle` 接口的类来管理生命周期。

### 6.1 Lifecycle 接口

Catalina 在设计上允许一个组件包含其他组件。例如，servlet 容器可以包含载入器、管理等。父组件负责启动 / 关闭它的子组件。Catalina 的这种设计使所有的组件都置于其父组件的“监护”之下，这样，Catalina 的启动类只需要启动一个组件就可以将全部应用的组件都启动起来。这种单一启动 / 关闭机制是通过 `Lifecycle` 接口实现的。代码清单 6-1 给出了 `Lifecycle` 接口的定义。

代码清单 6-1 Lifecycle 接口

```
package org.apache.catalina;

public interface Lifecycle {

    public static final String START_EVENT = "start";
    public static final String BEFORE_START_EVENT = "before_start";
    public static final String AFTER_START_EVENT = "after_start";
    public static final String STOP_EVENT = "stop";
    public static final String BEFORE_STOP_EVENT = "before_stop";
    public static final String AFTER_STOP_EVENT = "after_stop";

    public void addLifecycleListener(LifecycleListener listener);
```



```

public LifecycleListener[] findLifecycleListeners();
public void removeLifecycleListener(LifecycleListener listener);
public void start() throws LifecycleException;
public void stop() throws LifecycleException;
}

```

Lifecycle 接口中最重要的是 start() 方法和 stop() 方法。组件必须提供这两个方法的实现，供其父组件调用，以实现对其进行启动 / 关闭操作。其他的 3 个方法——addLifecycleListener()、findLifecycleListeners() 和 removeLifecycleListener()——都是与事件监听器相关的。组件中可以注册多个事件监听器来对发生在该组件上的某些事件进行监听。当某个事件发生时，相应的事件监听器会收到通知。Lifecycle 实例可以触发的 6 个事件都是以 Lifecycle 接口的公共静态 final 字符串表示。

## 6.2 LifecycleEvent 类

生命周期事件是 org.apache.catalina.LifecycleEvent 类的实例。代码清单 6-2 给出了该类的定义。

代码清单 6-2 LifecycleEvent 类的定义

```

package org.apache.catalina;
import java.util.EventObject;

public final class LifecycleEvent extends EventObject {
    public LifecycleEvent(Lifecycle lifecycle, String type) {
        this(lifecycle, type, null);
    }
    public LifecycleEvent(Lifecycle lifecycle, String type,
        Object data) {
        super(lifecycle);
        this.lifecycle = lifecycle;
        this.type = type;
        this.data = data;
    }
    private Object data = null;
    private Lifecycle lifecycle = null;
    private String type = null;

    public Object getData() {
        return (this.data);
    }
    public Lifecycle getLifecycle() {
        return (this.lifecycle);
    }
    public String getType() {
        return (this.type);
    }
}

```

## 6.3 LifecycleListener 接口

生命周期的事件监听器是 org.apache.catalina.LifecycleListener 接口的实例。代码清单 6-3 给

出了该接口的定义。

代码清单 6-3 LifecycleListener 类的定义

```
package org.apache.catalina;  
import java.util.EventObject;  
public interface LifecycleListener {  
    public void lifecycleEvent(LifecycleEvent event);  
}
```

该接口中只有一个方法，即 lifecycleEvent() 方法。当某个事件监听器监听到相关事件发生时，会调用该方法。

## 6.4 LifecycleSupport 类

实现了 Lifecycle 接口，并且对某个事件注册了监听器的组件必须提供 Lifecycle 接口中 3 个与监听器相关的方法（分别是 addLifecycleListener()、findLifecycleListeners() 和 removeLifecycleListener()）的实现。然后，该组件需要将所有注册的事件监听器存储到一个数组、ArrayList 或其他类似的对象中。Catalina 提供了一个工具类——org.apache.catalina.util.LifecycleSupport——来帮助组件管理监听器，并触发相应的生命周期事件。代码清单 6-4 给出了 LifecycleSupport 类的定义。

代码清单 6-4 LifecycleSupport 类的定义

```
package org.apache.catalina.util;  
import org.apache.catalina.Lifecycle;  
import org.apache.catalina.LifecycleEvent;  
import org.apache.catalina.LifecycleListener;  
  
public final class LifecycleSupport {  
    public LifecycleSupport(Lifecycle lifecycle) {  
        super();  
        this.lifecycle = lifecycle;  
    }  
  
    private Lifecycle lifecycle = null;  
    private LifecycleListener listeners[] = new LifecycleListener[0];  
    public void addLifecycleListener(LifecycleListener listener) {  
        synchronized (listeners) {  
            LifecycleListener results[] =  
                new LifecycleListener[listeners.length + 1];  
            for (int i = 0; i < listeners.length; i++)  
                results[i] = listeners[i];  
            results[listeners.length] = listener;  
            listeners = results;  
        }  
    }  
  
    public LifecycleListener[] findLifecycleListeners() {  
        return listeners;  
    }  
  
    public void fireLifecycleEvent(String type, Object data) {  
        LifecycleEvent event = new LifecycleEvent(lifecycle, type, data);  
        LifecycleListener interested[] = null;
```

```

synchronized (listeners) {
    interested = (LifecycleListener[]) listeners.clone();
}
for (int i = 0; i < interested.length; i++)
    interested[i].lifecycleEvent(event);
}

public void removeLifecycleListener(LifecycleListener listener) {
    synchronized (listeners) {
        int n = -1;
        for (int i = 0; i < listeners.length; i++) {
            if (listeners[i] == listener) {
                n = i;
                break;
            }
        }
        if (n < 0)
            return;
        LifecycleListener results[] =
            new LifecycleListener[listeners.length - 1];
        int j = 0;
        for (int i = 0; i < listeners.length; i++) {
            if (i != n)
                results[j++] = listeners[i];
        }
        listeners = results;
    }
}
}

```

正如代码清单 6-4 所示, LifecycleSupport 类将所有生命周期监听器存储在一个名为 listeners 的数组中, 并初始化为一个没有元素的数组对象:

```
private LifecycleListener listeners[] = new LifecycleListener[0];
```

当调用 addLifecycleListener() 方法添加一个事件监听器时, 会创建一个新数组, 大小为原数组的元素个数再加 1。然后, 将原数组中的所有元素复制到新数组中, 并将新的事件监听器添加到新数组中。当调用 removeLifecycleListener() 方法移除一个事件监听器时, 也会新建一个数组, 大小为原数组的元素个数再减 1, 然后将除了指定监听器外的其他所有监听器都复制到新数组中。

fireLifecycleEvent() 方法会触发一个生命周期事件。首先, 它会复制事件监听器数组。然后, 它调用数组中每个成员的 lifecycleEvent() 方法, 并传入要触发的事件。

实现 Lifecycle 接口的组件可以使用 LifecycleSupport 类。例如, 本章应用程序中的 SimpleContext 类声明了如下变量:

```
protected LifecycleSupport lifecycle = new LifecycleSupport(this);
```

当要添加一个事件监听器时, SimpleContext 实例会调用 LifecycleSupport 类的 addLifecycleListener() 方法:

```

public void addLifecycleListener(LifecycleListener listener) {
    lifecycle.addLifecycleListener(listener);
}

```



类似地，要移除某个事件监听器时，会调用 LifecycleSupport 类的 removeLifecycleListener() 方法：

```
public void removeLifecycleListener(LifecycleListener listener) {  
    lifecycle.removeLifecycleListener(listener);  
}
```

当要触发某个事件时，SimpleContext 类会调用 LifecycleSupport 类的 fireLifecycleEvent() 方法，如下所示：

```
lifecycle.fireLifecycleEvent(START_EVENT, null);
```

## 6.5 应用程序

本章的应用程序建立于第 5 章的应用程序基础之上，主要用来说明 Lifecycle 接口及生命周期相关类的使用方法。应用程序包含一个 Context 实例、两个 Wrapper 实例，还包括一个载入器和一个映射器。应用程序中的组件实现了 Lifecycle 接口，Context 实例中使用了事件监听器。为了使应用程序简单一些，取消了第 5 章中的两个阀。图 6-1 给出了应用程序中各相关类的 UML 类图。注意，有些类（SimpleContextValve、SimpleContextMapper 和 SimpleWrapperValve）和接口（Container、Wrapper、Context、Loader 和 Mapper）并没有显示在类图中。

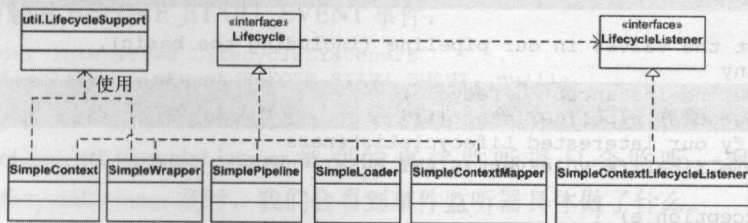


图 6-1 应用程序相关类的类图

注意，SimpleContext 类使用 SimpleContextLifecycleListener 类的实例作为事件监听器。SimpleContextValve 类、SimpleContextMapper 类和 SimpleWrapperValve 类与第 5 章中的类相同，这里不再讨论。

### 6.5.1 ex06.pyrmont.core.SimpleContext 类

本章应用程序中的 SimpleContext 类与第 5 章中的类类似，区别在于本章应用程序中的 SimpleContext 类实现了 Lifecycle 接口。SimpleContext 类使用变量 lifecycle 引用一个 LifecycleSupport 实例：

```
protected LifecycleSupport lifecycle = new LifecycleSupport(this);
```

此外，它还使用了一个布尔变量 started 来指明 SimpleContext 实例是否已经启动了。代码清单 6-5 给出了 SimpleContext 类提供的 Lifecycle 接口中方法的实现。

代码清单 6-5 Lifecycle 接口中方法的实现

```

public void addLifecycleListener(LifecycleListener listener) {
    lifecycle.addLifecycleListener(listener);
}

public LifecycleListener[] findLifecycleListeners() {
    return null;
}

public void removeLifecycleListener(LifecycleListener listener) {
    lifecycle.removeLifecycleListener(listener);
}

public synchronized void start() throws LifecycleException {
    if (started)
        throw new LifecycleException("SimpleContext has already started");
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);
    started = true;
    try {
        // Start our subordinate components, if any
        if ((loader != null) && (loader instanceof Lifecycle))
            ((Lifecycle) loader).start();

        // Start our child containers, if any
        Container children[] = findChildren();
        for (int i = 0; i < children.length; i++) {
            if (children[i] instanceof Lifecycle)
                ((Lifecycle) children[i]).start();
        }

        // Start the Valves in our pipeline (including the basic),
        // if any
        if (pipeline instanceof Lifecycle)
            ((Lifecycle) pipeline).start();
        // Notify our interested LifecycleListeners
        lifecycle.fireLifecycleEvent(START_EVENT, null);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(AFTER_START_EVENT, null);
}

public void stop() throws LifecycleException {
    if (!started)
        throw new LifecycleException("SimpleContext has not been started");
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(BEFORE_STOP_EVENT, null);
    lifecycle.fireLifecycleEvent(STOP_EVENT, null);
    started = false;
    try {
        // Stop the Valves in our pipeline (including the basic), if any
        if (pipeline instanceof Lifecycle) {
            ((Lifecycle) pipeline).stop();
        }

        // Stop our child containers, if any
        Container children[] = findChildren();
        for (int i = 0; i < children.length; i++) {
            if (children[i] instanceof Lifecycle)
                ((Lifecycle) children[i]).stop();
        }
        if ((loader != null) && (loader instanceof Lifecycle)) {

```

```

        ((Lifecycle) loader).stop();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(AFTER_STOP_EVENT, null);
}

```

需要注意的是，start() 方法是如何将所有子容器，以及与之相关的组件，包括载入器、管道和映射器等，启动起来的？另外又是如何关闭这些容器和组件的？答案就是使用前面所说的单一启动/关闭机制。使用这种机制，只需要启动最高层级的组件即可（在本例中是 SimpleContext），其余的组件会由各自的父组件去启动。同样，关闭这些组件时，也只需要关闭最高层级的组件即可。

SimpleContext 类的 start() 方法首先会检查组件之前是否已经启动过了，若是，则它抛出 LifecycleException 异常：

```

if (started)
    throw new LifecycleException(
        "SimpleContext has already started");

```

然后，它会触发 BEFORE\_START\_EVENT 事件：

```

// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);

```

这样，SimpleContext 实例中对该事件进行监听的所有监听器都会收到通知。在应用程序中，SimpleContextLifecycleListener 类型的事件监听器只会监听一种事件。在讨论 SimpleContextLifecycleListener 类时，我们会看到事件监听器具体做了什么。

接下来，start() 方法将布尔变量 started 设置为 true，表明该组件已经启动了：

```

started = true;

```

然后，start() 方法会启动它的组件和子容器。当前在应用程序中，共有两个组件实现了 Lifecycle 接口，分别是 SimpleLoader 类和 SimplePipeline 类。SimpleContext 类有两个 Wrapper 实例作为其子容器，它们都是 SimpleWrapper 类的实例，也都实现了 Lifecycle 接口：

```

try {
    // Start our subordinate components, if any
    if ((loader != null) && (loader instanceof Lifecycle))
        ((Lifecycle) loader).start();

    // Start our child containers, if any
    Container children[] = findChildren();
    for (int i = 0; i < children.length; i++) {
        if (children[i] instanceof Lifecycle)
            ((Lifecycle) children[i]).start();
    }

    // Start the Valves in our pipeline (including the basic),
    // if any
    if (pipeline instanceof Lifecycle)
        ((Lifecycle) pipeline).start();
}

```



在组件和子容器都启动完毕后，start() 方法会触发两个事件，START\_EVENT 和 AFTER\_START\_EVENT:

```
// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(START_EVENT, null);
.
.
.
// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(AFTER_START_EVENT, null);
```

stop() 方法首先会检查当前实例是否是启动的，若未启动，它会抛出 LifecycleException 异常:

```
if (!started)
    throw new LifecycleException(
        "SimpleContext has not been started");
```

然后，它会触发 BEFORE\_STOP\_EVENT 事件和 STOP\_EVENT 事件，重置布尔变量 started:

```
// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(BEFORE_STOP_EVENT, null);
lifecycle.fireLifecycleEvent(STOP_EVENT, null);
started = false;
```

接下来，stop() 方法会关闭与它关联的所有组件及 SimpleContext 实例的子容器:

```
try {
    // Stop the Valves in our pipeline (including the basic), if any
    if (pipeline instanceof Lifecycle) {
        ((Lifecycle) pipeline).stop();
    }

    // Stop our child containers, if any
    Container children[] = findChildren();
    for (int i = 0; i < children.length; i++) {
        if (children[i] instanceof Lifecycle)
            ((Lifecycle) children[i]).stop();
    }
    if ((loader != null) && (loader instanceof Lifecycle)) {
        ((Lifecycle) loader).stop();
    }
}
```

最后，它会触发 AFTER\_STOP\_EVENT 事件:

```
// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(AFTER_STOP_EVENT, null);
```

### 6.5.2 ex06.pyrmont.core.SimpleContextLifecycleListener 类

SimpleContext 实例中的事件监听器是 SimpleContextLifecycleListener 类的实例。代码清单 6-6 给出了 SimpleContextLifecycleListener 类的定义。

代码清单 6-6 SimpleContextLifecycleListener 类的定义

```
package ex06.pyrmont.core;
import org.apache.catalina.Context;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleEvent;
import org.apache.catalina.LifecycleListener;
```

```

public class SimpleContextLifecycleListener implements
LifecycleListener {

    public void lifecycleEvent(LifecycleEvent event) {
        Lifecycle lifecycle = event.getLifecycle();
        System.out.println("SimpleContextLifecycleListener's event " +
            event.getType().toString());
        if (Lifecycle.START_EVENT.equals(event.getType())) {
            System.out.println("Starting context.");
        }
        else if (Lifecycle.STOP_EVENT.equals(event.getType())) {
            System.out.println("Stopping context.");
        }
    }
}

```

SimpleContextLifecycleListener 类中 lifecycleEvent() 方法的实现比较简单，它仅仅会输出已触发事件类型。若该事件是 START\_EVENT，则 lifecycleEvent() 方法还会输出字符串“Starting context.”；如果事件是 STOP\_EVENT，则它输出字符串“Stopping context.”。

### 6.5.3 ex06.pyrmont.core.SimpleLoader 类

这里的 SimpleLoader 类与第 5 章中的类类似，区别在于在该应用程序中 SimpleLoader 类实现了 Lifecycle 接口。这个类的 Lifecycle 接口中各个方法的实现只是向控制台中输出字符串。但重要的是，通过实现 Lifecycle 接口，启动 SimpleLoader 实例的任务就可以由与其相关联的 servlet 容器来完成。

代码清单 6-7 给出了 Lifecycle 接口里声明的方法在 SimpleLoader 类中的实现。

代码清单 6-7 Lifecycle 接口里声明的方法在 SimpleLoader 类中的实现

```

public void addLifecycleListener(LifecycleListener listener) {}
public LifecycleListener[] findLifecycleListeners() {
    return null;
}
public void removeLifecycleListener(LifecycleListener listener) {}
public synchronized void start() throws LifecycleException {
    System.out.println("Starting SimpleLoader");
}
public void stop() throws LifecycleException {}

```

### 6.5.4 ex06.pyrmont.core.SimplePipeline 类

除了实现 Pipeline 接口外，SimplePipeline 类还实现了 Lifecycle 接口。Lifecycle 接口中声明的方法这里都留空了，但是 SimplePipeline 类的实例已经可以由与其相关联的 servlet 容器来启动。该类中剩下的内容与第 5 章中的 SimplePipeline 类类似。

### 6.5.5 ex06.pyrmont.core.SimpleWrapper 类

该类与 ex05.pyrmont.core.SimpleWrapper 类相似。在本章的应用程序中，该类还实现了

Lifecycle 接口，注意，就可以由其父容器来启动该实例。在该应用程序中，Lifecycle 接口里声明的大部分方法都留空了，只有 start() 方法和 stop() 方法给出了实现。代码清单 6-8 给出了 Lifecycle 接口中相关方法的实现。

代码清单 6-8 Lifecycle 接口中相关方法的实现

```

public void addLifecycleListener(LifecycleListener listener) { }
public LifecycleListener[] findLifecycleListeners() {
    return null;
}
public void removeLifecycleListener(LifecycleListener listener) { }
public synchronized void start() throws LifecycleException {
    System.out.println("Starting Wrapper " + name);
    if (started)
        throw new LifecycleException("Wrapper already started");
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);
    started = true;

    // Start our subordinate components, if any
    if ((loader != null) && (loader instanceof Lifecycle))
        ((Lifecycle) loader).start();
    // Start the Valves in our pipeline (including the basic), if any
    if (pipeline instanceof Lifecycle)
        ((Lifecycle) pipeline).start();
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(START_EVENT, null);
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(AFTER_START_EVENT, null);
}

public void stop() throws LifecycleException {
    System.out.println("Stopping wrapper " + name);
    // Shut down our servlet instance (if it has been initialized)
    try {
        instance.destroy();
    }
    catch (Throwable t) {
    }
    instance = null;
    if (!started)
        throw new LifecycleException("Wrapper " + name + " not started");
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(BEFORE_STOP_EVENT, null);
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(STOP_EVENT, null);
    started = false;

    // Stop the Valves in our pipeline (including the basic), if any
    if (pipeline instanceof Lifecycle) {
        ((Lifecycle) pipeline).stop();
    }
    // Stop our subordinate components, if any
    if ((loader != null) && (loader instanceof Lifecycle)) {
        ((Lifecycle) loader).stop();
    }
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(AFTER_STOP_EVENT, null);
}

```



SimpleWrapper 类的 start() 方法与 SimpleContext 类的 start() 方法类似。它会启动添加到其中的所有组件（在本应用程序中，没有需要由它启动的组件），并触发 BEFORE\_START\_EVENT、START\_EVENT 和 AFTER\_START\_EVENT 事件。

SimpleWrapper 类中的 stop() 方法比较有趣。除了输出一个简单的字符串外，它还要调用 servlet 实例的 destroy() 方法：

```
System.out.println("Stopping wrapper " + name);
// Shut down our servlet instance (if it has been initialized)
try {
    instance.destroy();
}
catch (Throwable t) {
}
instance = null;
```

然后，它会检查 Wrapper 实例是否已启动的，若不是，则它抛出 LifecycleException 异常：

```
if (!started)
    throw new LifecycleException("Wrapper " + name + " not started");
```

接下来，它会触发 BEFORE\_STOP\_EVENT 和 STOP\_EVENT 事件，并重置布尔变量 started：

```
// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(BEFORE_STOP_EVENT, null);
// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(STOP_EVENT, null);
started = false;
```

然后，它会关闭与其相关联的载入器和管道组件。在本应用程序中，SimpleWrapper 实例并没有载入器：

```
// Stop the Valves in our pipeline (including the basic), if any
if (pipeline instanceof Lifecycle) {
    ((Lifecycle) pipeline).stop();
}
// Stop our subordinate components, if any
if ((loader != null) && (loader instanceof Lifecycle)) {
    ((Lifecycle) loader).stop();
}
```

最后，它触发 AFTER\_STOP\_EVENT 事件：

```
// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(AFTER_STOP_EVENT, null);
```

## 6.5.6 运行应用程序

要在 Windows 平台下运行该应用程序，需要在工作目录下执行如下命令：

```
java -classpath ./lib/servlet.jar;./ ex06.pyrmont.startup.Bootstrap
```

要在 Linux 平台下运行，需要使用冒号来代替库文件之间的分号：

```
java -classpath ./lib/servlet.jar:./ ex06.pyrmont.startup.Bootstrap
```

然后，你会在控制台中看到如下消息。请注意不同的事件是如何触发的。

```
HttpConnector Opening server socket on all host IP addresses
HttpConnector[8080] Starting background thread
```

```

SimpleContextLifecycleListener's event before_start
Starting SimpleLoader
Starting Wrapper Modern
Starting Wrapper Primitive
SimpleContextLifecycleListener's event start
Starting context.
SimpleContextLifecycleListener's event after_start

```

要想调用 `Primitiveservlet`，可以在浏览器中使用如下的 URL：

```
http://localhost:8080/Primitive
```

类似地，要调用 `ModernServlet` 时，可以使用如下的 URL：

```
http://localhost:8080/Modern
```

## 6.6 小结

在本章中，你已经学会了如何使用 `Lifecycle` 接口来管理组件。该接口定义了组件的生命周期，并提供了一种优雅的方式向其他的组件发送事件消息。此外，通过使用 `Lifecycle` 接口，`Catalina` 就可以使用单一启动/关闭机制来启动/关闭所有的组件。

# 第 7 章

## 日志记录器

日志记录器是用来记录消息的组件。在 Catalina 中，日志记录器需要与某个 servlet 容器相关联，与其他组件比，相对简单一些。在 org.apache.catalina.logger 包下，Tomcat 提供了几种不同类型的日志记录器。本章的应用程序位于 ex07.pyrmont 包下。与第 6 章的应用程序相比，只有两个类发生了变化，SimpleContext 类和 Bootstrap 类。

本章内容共分为 3 节。第 1 节将对 org.apache.catalina.Logger 接口进行介绍，Tomcat 中所有的日志记录器都必须实现该接口，第 2 节将介绍 Tomcat 中的日志记录器，第 3 节将对本章使用 Tomcat 的应用程序中的日志记录器进行讨论。

### 7.1 Logger 接口

Tomcat 中的日志记录器都必须实现 org.apache.catalina.Logger 接口。org.apache.catalina.Logger 接口的定义在代码清单 7-1 中给出。

代码清单 7-1 Logger 接口的定义

```
package org.apache.catalina;
import java.beans.PropertyChangeListener;

public interface Logger {
    public static final int FATAL = Integer.MIN_VALUE;
    public static final int ERROR = 1;
    public static final int WARNING = 2;
    public static final int INFORMATION = 3;
    public static final int DEBUG = 4;

    public Container getContainer();
    public void setContainer(Container container);
    public String getInfo();
    public int getVerbosity();
    public void setVerbosity(int verbosity);
    public void addPropertyChangeListener(PropertyChangeListener listener);
    public void log(String message);
    public void log(Exception exception, String msg);
    public void log(String message, Throwable throwable);
    public void log(String message, int verbosity);
    public void log(String message, Throwable throwable, int verbosity);
    public void removePropertyChangeListener(PropertyChangeListener listener);
}
```

Logger 接口提供了一些 log() 方法来写日志，其实现类可以选择性地调用。其中最简单的方法



法是只作为消息接受一个要记录的字符串。

Logger 接口的最后两个 log() 方法接受一个日志级别参数。如果参数的日志级别的数字比该日志记录器实例中设定的等级低，才会记录该条消息；否则，会忽略该条记录。Logger 接口中共定义了 5 种日志级别，分别是 FATAL、ERROR、WARNING、INFORMATION 和 DEBUG。可以使用 getVerbosity() 方法和 setVerbosity() 方法来获取 / 设置日志级别。

此外，可以通过 Logger 接口的 getContainer() 方法和 setContainer() 方法将日志记录器和某个 servlet 容器相关联。addPropertyChangeListener() 方法和 removePropertyChangeListener() 方法可以用来添加 / 移除 PropertyChangeListener 的实例。

## 7.2 Tomcat 的日志记录器

Tomcat 提供了 3 种日志记录器，其类分别是 FileLogger、SystemErrorLogger 和 SystemOutLogger。这 3 个类都位于 org.apache.catalina.logger 包下，均继承自 org.apache.catalina.logger.LoggerBase 类。在 Tomcat 4 中 LoggerBase 类实现了 org.apache.catalina.Logger 接口。在 Tomcat 5 中，LoggerBase 类还实现了 Lifecycle 接口（参见第 6 章）和 MBeanRegistration 接口（参见第 20 章）。图 7-1 给出了日志记录器相关类的 UML 类图。

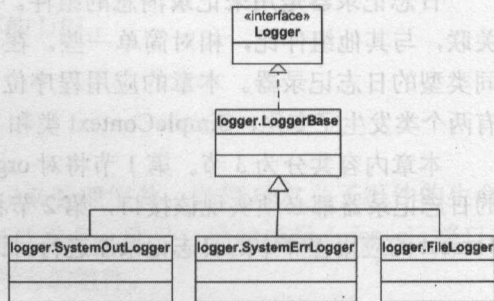


图 7-1 Tomcat 中的日志记录器相关类的 UML 类图

### 7.2.1 LoggerBase 类

在 Tomcat 5 中，LoggerBase 类会更复杂一些，因为它与创建 Mbeans 接口（仅在第 20 章讨论）的代码进行了整合。因此，这里只讨论 Tomcat 4 中的 LoggerBase 类。当你理解了第 20 章的内容后，就能理解 Tomcat 5 中 LoggerBase 类的实现原理。

在 Tomcat 4 中，LoggerBase 是一个抽象类，它实现了 Logger 接口中除 log(String msg) 方法外的全部方法的实现。需要子类实现的方法签名如下：

```
public abstract void log(String msg);
```

该方法在其子类中给出具体的实现，并有多种重载方法，所有的重载 log() 方法最终都会调用该方法。由于子类中的重载方法都有不同的实现目的，因此在 LoggerBase 类中，该方法是留空的。

现在来看下 LoggerBase 类的日志等级设置。日志等级是使用一个名为 verbosity 的变量来设置的，默认值为 ERROR：

```
protected int verbosity = ERROR;
```

变量 verbosity 的值可以使用 setVerbosity() 方法来设置，可使用的字符串包括：FATAL、ERROR、WARNING、INFORMATION 或 DEBUG。代码清单 7-2 给出了 LoggerBase 类的 setVerbosity() 方法的实现。

---

**代码清单 7-2 LoggerBase 类的 setVerbosity 方法**

---

```
public void setVerbosityLevel(String verbosity) {  
    if ("FATAL".equalsIgnoreCase(verbosity))  
        this.verbosity = FATAL;  
    else if ("ERROR".equalsIgnoreCase(verbosity))  
        this.verbosity = ERROR;  
    else if ("WARNING".equalsIgnoreCase(verbosity))  
        this.verbosity = WARNING;  
    else if ("INFORMATION".equalsIgnoreCase(verbosity))  
        this.verbosity = INFORMATION;  
    else if ("DEBUG".equalsIgnoreCase(verbosity))  
        this.verbosity = DEBUG;  
}
```

---

代码清单 7-3 中的两个 log() 方法都会接受一个整型参数来标明日志等级。在这两个重载的方法中,只有当传入的日志等级比当前实例中 verbosity 变量指定的等级低时,才会调用重载方法 log(String message) 来记录日志。

---

**代码清单 7-3 接受日志等级的两个 log() 重载方法**

---

```
public void log(String message, int verbosity) {  
    if (this.verbosity >= verbosity)  
        log(message);  
}  
public void log(String message, Throwable throwable, int verbosity) {  
    if (this.verbosity >= verbosity)  
        log(message, throwable);  
}
```

---

下面几节会讨论 LoggerBase 的 3 个子类,其中会看到重载的 log(String message) 方法的实现。

## 7.2.2 SystemOutLogger 类

该类继承自 LoggerBase,提供了 log(String message) 重载方法的重载实现。接收到的每条日志消息都会传递给 System.out.println() 方法。代码清单 7-4 给出了 SystemOutLogger 类的定义。

---

**代码清单 7-4 SystemOutLogger 类的定义**

---

```
package org.apache.catalina.logger;  
public class SystemOutLogger extends LoggerBase {  
  
    protected static final String info =  
        "org.apache.catalina.logger.SystemOutLogger/1.0";  
    public void log(String msg) {  
        System.out.println(msg);  
    }  
}
```

---

## 7.2.3 SystemErrLogger 类

该类与 SystemOutLogger 类似,区别在于该类的 log(String message) 重载方法会调用 System.err.println() 方法,将日志信息输出到标准错误。代码清单 7-5 给出了 SystemErrLogger 类的定义。

代码清单 7-5 SystemErrLogger 类的定义

```
package org.apache.catalina.logger;  
public class SystemErrLogger extends LoggerBase {  
  
    protected static final String info =  
        "org.apache.catalina.logger.SystemErrLogger/1.0";  
    public void log(String msg) {  
        System.err.println(msg);  
    }  
}
```

## 7.2.4 FileLogger 类

FileLogger 类是 LoggerBase 类的 3 个子类中最复杂的一个。它会将从 servlet 容器中接收到的日志消息写到一个文件中，并且可以选择是否要为每条消息添加时间戳。当该类首次被实例化时，会创建一个日志文件，文件名包含当日的日期信息。若日期发生了变化，则创建一个新文件，并将所有的日志消息都写到新文件中。使用该类的实例时，可以在日志文件的名称中添加前缀和后缀。

在 Tomcat 4 中，FileLogger 类实现 Lifecycle 接口，这样就可以像其他实现了 org.apache.catalina.Lifecycle 接口的组件一样由相关联的 servlet 容器负责启动/关闭。在 Tomcat 5 中，改为由 LoggerBase 类（FileLogger 的父类）实现 Lifecycle 接口。

在 Tomcat 4 中，LoggerBase 类的 start() 方法和 stop() 方法（继承自 Lifecycle 接口）只负责在启动/关闭文件日志记录器时触发生命周期事件，并不做其他的事情。代码清单 7-6 给出了这两个方法的实现。注意，stop() 方法还会调用其私有方法 close() 来关闭打开的日志文件。close() 方法将在本节后面讨论。

代码清单 7-6 start() 方法和 stop() 方法的实现

```
public void start() throws LifecycleException {  
    // Validate and update our current component state  
    if (started)  
        throw new LifecycleException  
            (sm.getString("fileLogger.alreadyStarted"));  
    lifecycle.fireLifecycleEvent(START_EVENT, null);  
    started = true;  
}  
  
public void stop() throws LifecycleException {  
    // Validate and update our current component state  
    if (!started)  
        throw new LifecycleException  
            (sm.getString("fileLogger.notStarted"));  
    lifecycle.fireLifecycleEvent(STOP_EVENT, null);  
    started = false;  
    close();  
}
```

FileLogger 类中最重要的方法是 log() 方法，其具体实现在代码清单 7-7 中给出。



代码清单 7-7 log() 方法的实现

```
public void log(String msg) {
    // Construct the timestamp we will use, if requested
    Timestamp ts = new Timestamp(System.currentTimeMillis());
    String tsString = ts.toString().substring(0, 19);
    String tsDate = tsString.substring(0, 10);

    // If the date has changed, switch log files
    if (!date.equals(tsDate)) {
        synchronized (this) {
            if (!date.equals(tsDate)) {
                close();
                date = tsDate;
                open();
            }
        }
    }

    // Log this message, timestamped if necessary
    if (writer != null) {
        if (timestamp) {
            writer.println(tsString + " " + msg);
        }
        else {
            writer.println(msg);
        }
    }
}
```

log() 方法会将接收到的日志消息写到一个日志文件中。在 FileLogger 实例的整个生命周期中，log() 方法可能会打开 / 关闭多个日志文件。典型情况下，当日期发生变化时，log() 方法会关闭当前日志文件，并打开一个新文件。下面来看一下 open()、close() 和 log() 方法的具体实现。

#### 1. open() 方法

代码清单 7-8 给出了 open() 方法的具体实现。它会在指定目录创建一个新的日志文件。

代码清单 7-8 open() 方法的实现

```
private void open() {
    // Create the directory if necessary
    File dir = new File(directory);
    if (!dir.isDirectory())
        dir = new File(System.getProperty("catalina.base"), directory);
    dir.mkdirs();

    // Open the current log file
    try {
        String pathname = dir.getAbsolutePath() + File.separator +
            prefix + date + suffix;
        writer = new PrintWriter(new FileWriter(pathname, true), true);
    }
    catch (IOException e) {
        writer = null;
    }
}
```

open() 方法首先会检查要创建的日志文件所在的目录是否存在，如果该目录不存在，则

open() 方法创建该目录。目录位置是存储在类变量 directory 中的:

```
File dir = new File(directory);
if (!dir.isAbsolute())
    dir = new File(System.getProperty("catalina.base"), directory);
dir.mkdirs();
```

然后, 根据待打开日志文件的目录路径, 创建位于指定位置的日志文件, 并添加相应的前缀、当前日期和后缀:

```
try {
    String pathname = dir.getAbsolutePath() + File.separator +
        prefix + date + suffix;
```

接下来, 它会创建一个 java.io.PrintWriter 实例, 具体执行写路径名操作的是 java.io.FileWriter 对象的实例。然后, 将 PrintWriter 实例赋值给类变量 writer。log() 方法使用变量 writer 来记录日志消息。

```
writer = new PrintWriter(new FileWriter(pathname, true), true);
```

## 2. close() 方法

close() 方法负责确保将 PrintWriter 实例中所有的日志消息都写到文件中, 关闭 PrintWriter 实例, 将其引用置为 null, 并将日期字符串清空。代码清单 7-9 给出了 close() 方法的实现。

代码清单 7-9 close() 方法的实现

```
private void close() {
    if (writer == null)
        return;
    writer.flush();
    writer.close();
    writer = null;
    date = "";
}
```

## 3. log() 方法

log() 方法会先创建 java.sql.Timestamp 类的一个实例 (该类是 java.util.Date 类的一个瘦包装)。实例化 Timestamp 类旨在方便获取当前日期。log() 方法是通过传入一个以长整数表示的当前日期到 Timestamp 类的构造函数中来创建 Timestamp 实例的:

```
Timestamp ts = new Timestamp(System.currentTimeMillis());
```

使用 Timestamp 类的 toString() 方法可以获取当前日期的字符串表示形式。toString() 方法返回的日期是的格式如下:

```
yyyy-mm-dd hh:mm:ss.fffffffff
```

其中 ffffffff 表示从 00:00:00 起经过的纳秒数。若是只想获取日期和小时, 则可以调用 String 类的 substring() 方法, 如下所示:

```
String tsString = ts.toString().substring(0, 19);
```

然后, 对于变量 tsString, 可以使用下面的代码来获取日期部分:

```
String tsDate = tsString.substring(0, 10);
```

然后, `log()` 方法将 `tsDate` 与字符串形式的变量 `date` (该变量最初为空字符串) 相比较。若二者的值不相等, 则 `log()` 方法会关闭当前日志文件, 将 `tsDate` 的值赋给 `date`, 并打开一个新的日志文件:

```
// If the date has changed, switch log files
if (!date.equals(tsDate)) {
    synchronized (this) {
        if (!date.equals(tsDate)) {
            close();
            date = tsDate;
            open();
        }
    }
}
```

最后, `log()` 方法将日志信息写到 `PrintWriter` 实例中, `PrintWriter` 实例使用输出流写入到日志文件中。若布尔变量 `timestamp` 的值为 `true`, 则它会在日志消息前添加时间戳 (`ts String`)。否则, 它会记录不带前缀的消息。

```
// Log this message, timestamped if necessary
if (writer != null) {
    if (timestamp) {
        writer.println(tsString + " " + msg);
    }
    else {
        writer.println(msg);
    }
}
```

### 7.3 应用程序

本章的应用程序与第 6 章中的应用程序相似, 区别在于, `SimpleContext` 对象关联了一个 `FileLogger` 组件。第 6 章中应用程序的具体变化在 `ex07.pyrmont.startup.Bootstrap` 类的 `main()` 方法中, 其具体实现在代码清单 7-10 中给出。尤其要注意突出显示的代码。

代码清单 7-10 Bootstrap 类的定义

```
package ex07.pyrmont.startup;

import ex07.pyrmont.core.SimpleContext;
import ex07.pyrmont.core.SimpleContextLifecycleListener;
import ex07.pyrmont.core.SimpleContextMapper;
import ex07.pyrmont.core.SimpleLoader;
import ex07.pyrmont.core.SimpleWrapper;
import org.apache.catalina.Connector;
import org.apache.catalina.Context;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleListener;
import org.apache.catalina.Loader;
import org.apache.catalina.logger.FileLogger;
import org.apache.catalina.Mapper;
import org.apache.catalina Wrapper;
import org.apache.catalina.connector.http.HttpConnector;

public final class Bootstrap {
    public static void main(String[] args) {
```



```

Connector connector = new HttpConnector();
Wrapper wrapper1 = new SimpleWrapper();
wrapper1.setName("Primitive");
wrapper1.setServletClass("PrimitiveServlet");
Wrapper wrapper2 = new SimpleWrapper();
wrapper2.setName("Modern");
wrapper2.setServletClass("ModernServlet");
Loader loader = new SimpleLoader();

Context context = new SimpleContext();
context.addChild(wrapper1);
context.addChild(wrapper2);

Mapper mapper = new SimpleContextMapper();
mapper.setProtocol("http");
LifecycleListener listener = new SimpleContextLifecycleListener();
((Lifecycle) context).addLifecycleListener(listener);
context.addMapper(mapper);
context.setLoader(loader);
// context.addServletMapping(pattern, name);
context.addServletMapping("/Primitive", "Primitive");
context.addServletMapping("/Modern", "Modern");

```

```

// ----- add logger -----
System.setProperty("catalina.base",
System.getProperty("user.dir"));
FileLogger logger = new FileLogger();
logger.setPrefix("FileLog_");
logger.setSuffix(".txt");
logger.setTimestamp(true);
logger.setDirectory("webroot");
context.setLogger(logger);

```

```

//-----
connector.setContainer(context);
try {
    connector.initialize();
    ((Lifecycle) connector).start();
    ((Lifecycle) context).start();

    // make the application wait until we press a key.
    System.in.read();
    ((Lifecycle) context).stop();
}
catch (Exception e) {
    e.printStackTrace();
}
}
}

```

## 7.4 小结

在本章中，你已经学习了关于日志记录器组件的知识，熟悉了 `org.apache.catalina.Logger` 接口，并了解了 Tomcat 中 `Logger` 接口的 3 个实现类的详细内容。此外，在本章的应用程序中，使用 `FileLogger` 类（Tomcat 中最高级的日志记录器）来记录相关日志信息。

## 第 8 章

# 载 入 器

在前面的章节中，你已经见过了如何使用一个简单的载入器来载入所需要的 servlet 类。本章将要介绍一个标准 Web 应用程序中的载入器，简单来说就是 Tomcat 中的载入器。servlet 容器需要实现一个自定义的载入器，而不能使用简单地使用系统的类载入器，因为 servlet 容器不应该完全信任它正在运行的 servlet 类。如果像前几章一样，使用系统类的载入器载入某个 servlet 类所使用的全部类，那么 servlet 就能够访问所有的类，包括当前运行的 Java 虚拟机（Java Virtual Machine, JVM）中环境变量 CLASSPATH 指明的路径下的所有的类和库。这是非常危险的。servlet 应该只允许载入 WEB-INF/classes 目录及其子目录下的类，和从部署的库到 WEB-INF/lib 目录载入类。这就是为什么 servlet 容器需要实现一个自定义的载入器。载入器使用类载入器，后者应用某些规则载入类。在 Catalina 中，载入器是 org.apache.catalina.Loader 接口的实例。

Tomcat 中需要实现自定义载入器的另一个原因是，为了提供自动重载的功能，即当 WEB-INF/classes 目录或 WEB-INF/lib 目录下的类发生变化时，Web 应用程序会重新载入这些类。在 Tomcat 的载入器的实现中，类载入器使用一个额外的线程来不断地检查 servlet 类和其他类的文件的时间戳。若要支持自动重载功能，则载入器必须实现 org.apache.catalina.loader.Reloader 接口。

8.1 节会简要回顾 Java 中的类载入机制。接下来会对所有载入器都必须实现的 Loader 接口进行介绍，然后介绍 Reloader 接口的。在了解了载入器和类载入器的实现机制后，本章通过使用一个应用程序来说明如何使用 Tomcat 的载入器。

本章有两个术语需要注意：仓库（repository）和资源（resource）。仓库表示类载入器会在哪里搜索要载入的类，而资源指的是一个类载入器中的 DirContext 对象，它的文件根路径指的是上下文的文件根路径。

### 8.1 Java 的类载入器

每次创建 Java 类的实例时，都必须现将类载入到内存中。Java 虚拟机使用类载入器来载入需要的类。一般情况下，类载入器会在一些 Java 核心类库，以及环境变量 CLASSPATH 中指定的目录中搜索相关类。如果在这些位置它都找不到要载入的类，就会抛出 java.lang.ClassNotFoundException 异常。

从 J2SE 1.2 开始，JVM 使用了 3 种类载入器来载入所需要的类，分别是引导类载入器（bootstrap class loader）、扩展类载入器（extension class loader）和系统类载入器（system class loader）。3 种类载入器之间是父子继承关系，其中引导类载入器位于层次结构的最上层，系统类载入器位于最下层。

引导类加载器用于引导启动 Java 虚拟机。当调用 `javax.exe` 程序时，就会启动引导类加载器。引导类加载器是使用本地代码来实现的，因为它用来载入运行 JVM 所需要的类，以及所有的 Java 核心类，例如 `java.lang` 包和 `java.io` 包下的类。启动类加载器会在 `rt.jar` 和 `i18n.jar` 等 Java 包中搜索要载入的类。引导类加载器要从哪些库中搜索类依赖于 JVM 和操作系统的版本。

扩展类加载器负责载入标准扩展目录中的类。这有利于程序开发，因为程序员只需要将 JAR 文件复制到扩展目录中就可以被类加载器搜索到。扩展库依赖于 JDK 供应商的具体实现。Sun 公司的 JVM 的标准扩展目录是 `/jdk/jre/lib/ext`。

系统类加载器是默认类加载器，它会搜索在环境变量 `CLASSPATH` 中指明的路径和 JVR 文件。

那么，JVM 使用的是哪个类加载器呢？答案在于类加载器的代理模型。使用代理模型可以有效地解决类载入过程中的安全性问题。每当需要载入一个类的时候，会首先调用系统类加载器。但是，它并不会立即载入某个类。相反，它会将载入类的任务交给其父加载器，即扩展类加载器，而扩展类加载器又会将载入任务交给其父加载器，即引导类加载器。因此，引导类加载器总是会首先执行载入某个类的任务。如果引导类加载器找不到需要载入的类，那么扩展类加载器会尝试载入该类。如果扩展类加载器也找不到这个类，就轮到系统类加载器继续执行载入任务。如果系统类加载器还是找不到这个类，则会抛出 `java.lang.ClassNotFoundException` 异常。那么，为什么要执行这样一个循环过程呢？

代理模型的重要用途就是为了解决类载入过程中的安全性问题。如你所知，可以使用安全管理器来限制某个类对某个路径的访问。现在，某个恶意用户编写了一个名为 `java.lang.Object` 的类，它可以访问硬盘中的任意目录。由于 JVM 是信任 `java.lang.Object` 类的，这样，它就不会监视这个类的活动。结果是，如果这个自定义 `java.lang.Object` 允许载入，安全管理器就这样被轻易地绕过了。幸运的是，由于使用了代理模型，这种情况是不会发生的。下面来说说具体原因。

当程序中的某个地方调用了自定义 `java.lang.Object` 类时，系统类加载器会将载入任务委托给扩展类加载器，继而会被交给引导类加载器。引导类加载器搜索其核心库，找到标准的 `java.lang.Object` 类，并将之实例化。结果是，自定义 `java.lang.Object` 类并没有被载入。

关于 Java 中类载入机制的一件重要的事情是，可以通过继承抽象类 `java.lang.ClassLoader` 类编写自己的类加载器。而 Tomcat 要使用自定义类加载器的原因有以下 3 条：

- 为了在载入类中指定某些规则；
- 为了缓存已经载入的类；
- 为了实现类的预载入，方便使用。

## 8.2 Loader 接口

在载入 Web 应用程序中需要的 `servlet` 类及其相关类时要遵守一些明确的规则。例如，应用程序中的 `servlet` 只能引用部署在 `WEB-INF/classes` 目录及其子目录下的类。但是，`servlet` 类不能访问其他路径中的类，即使这些类包含在运行当前 Tomcat 的 JVM 的 `CLASSPATH` 环境变量中。此外，`servlet` 类只能访问 `WEB-INF/lib` 目录下的库，其他目录中的类库均不能访问。

Tomcat 中的载入器指的是 Web 应用程序载入器，而不仅仅是指类加载器。载入器必须实现



org.apache.catalina.Loader 接口。在载入器的实现中,会使用一个自定义类载入器,它是 org.apache.catalina.loader.WebappClassLoader 类的一个实例。可以使用 Loader 接口的 getClassLoader() 方法来获取 Web 载入器中 ClassLoader 类的实例。

此外,Loader 接口还定义了一些方法来对仓库的集合进行操作。Web 应用程序中的 WEB-INF/classes 目录和 WEB-INF/lib 目录是作为仓库添加到载入器中的。使用 Loader 接口的 addRepository() 方法可以添加一个新的仓库,而其 findRepositories() 方法则返回所有已添加的仓库的数组对象。

Tomcat 的载入器通常会与一个 Context 级别的 servlet 容器相关联,Loader 接口的 getContainer() 方法和 setContainer() 方法用来将载入器与某个 servlet 容器相关联。如果 Context 容器中的一个或多个类被修改了,载入器也可以支持对类的自动重载。这样,servlet 程序员就可以重新编译 servlet 类及其相关类,并将其重新载入而不需要重新启动 Tomcat。Loader 接口使用 modified() 方法来支持类的自动重载。在载入器的具体实现中,如果仓库中的一个或多个类被修改了,那么 modified() 方法必须返回 true,才能提供自动重载的支持。但是,载入器类本身并不能自动重载。相反,它会调用 Context 接口的 reload() 方法来实现。其他的两个相关方法, setReloadable() 方法和 getReloadable() 方法,用来指明是否支持载入器的自动重载。默认情况下,在 Context 接口的标准实现中(即 org.apache.catalina.core.StandardContext 类,将会在第 12 章中讨论),是禁用了自动重载功能的。因此,要想启用 Context 容器的自动重载功能,需要在 server.xml 文件中添加一个 Context 元素,如下所示:

```
<Context path="/myApp" docBase="myApp" debug="0" reloadable="true"/>
```

此外,载入器的实现会指明是否要委托给一个父类载入器。为此,Loader 接口中声明了 getDelegate() 方法和 setDelegate() 方法。

代码清单 8-1 给出了 Loader 接口的定义。

代码清单 8-1 Loader 接口的定义

```
package org.apache.catalina;  
import java.beans.PropertyChangeListener;
```

```
public interface Loader {  
    public ClassLoader getClassLoader();  
    public Container getContainer();  
    public void setContainer(Container container);  
    public DefaultContext getDefaultContext();  
    public void setDefaultContext(DefaultContext defaultContext);  
    public boolean getDelegate();  
    public void setDelegate(boolean delegate);  
    public String getInfo();  
    public boolean getReloadable();  
    public void setReloadable(boolean reloadable);  
    public void addPropertyChangeListener(PropertyChangeListener  
        listener);  
    public void addRepository(String repository);  
    public String[] findRepositories();  
    public boolean modified();  
    public void removePropertyChangeListener(PropertyChangeListener  
        listener);  
}
```

Catalina 提供了 `org.apache.catalina.loader.WebappLoader` 类作为 `Loader` 接口的实现。其中, `WebappLoader` 对象中使用 `org.apache.catalina.loader.WebappClassLoader` 类的实例作为其类载入器, 该类继承自 `java.net.URLClassLoader` 类。

**注意** 当与某个载入器相关联的容器需要使用某个 `Servlet` 类时, 即当该类的某个方法被调用时, 容器会先调用载入器的 `getClassLoader()` 方法来获取类载入器的实例。然后, 容器会调用类载入器的 `loadClass()` 方法来载入这个 `Servlet` 类。更多详细信息请参见第 11 章内容。

`Loader` 接口及其实现类的 UML 类图如图 8-1 所示。

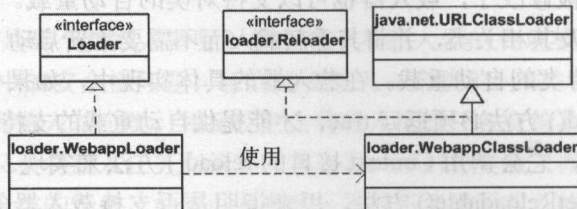


图 8-1 `Loader` 接口及其实现类的 UML 类图

### 8.3 Reloadable 接口

为了支持类的自动重载功能, 类载入器实现需要实现 `org.apache.catalina.loader.Reloadable` 接口。该接口的定义在代码清单 8-2 中给出。

代码清单 8-2 `Reloadable` 接口的定义

```

package org.apache.catalina.loader;
public interface Reloadable {
    public void addRepository(String repository);
    public String[] findRepositories();
    public boolean modified();
}
  
```

`Reloadable` 接口中最重要的方法是 `modified()` 方法。如果 Web 应用程序中的某个 `Servlet` 或相关类被修改了, `modified()` 方法会返回 `true`。 `addRepository()` 方法用来添加一个仓库, `findRepositories()` 方法返回一个字符串数组, 其中包含了实现 `Reloadable` 接口的类载入器的所有仓库。

### 8.4 WebappLoader 类

`org.apache.catalina.loader.WebappLoader` 类实现 `Loader` 接口, 其实例就是 Web 应用程序中的载入器, 负责载入 Web 应用程序中所使用到的类。 `WebappLoader` 类会创建 `org.apache.catalina.loader.WebappClassLoader` 类的一个实例作为其类载入器。像其他的 Catalina 组件一样, `WebappLoader` 类也实现了 `org.apache.catalina.Lifecycle` 接口, 可以由其相关联的容器来启动或关闭。此外, `WebappLoader` 类还实现 `java.lang.Runnable` 接口, 这样, 它就可以指定一个线程来不断地调用其类载入器的 `modified()` 方法。如果 `modified()` 方法返回 `true`, `WebappLoader` 的实例会通知其关联的 `Servlet` 容器 (在这里是 `Context` 类的实例)。然后由 `Context` 实例而不是

WebappLoader 实例，来完成类的重新载入。Context 如何完成这项工作将在第 12 章讨论。

当调用 WebappLoader 类的 start() 方法时，会完成以下几项重要工作：

- 创建一个类载入器；
- 设置仓库；
- 设置类路径；
- 设置访问权限；
- 启动一个新线程来支持自动重载。

这些任务将在下面几节中分别讨论。

### 8.4.1 创建类载入器

为了完成载入类的任务，WebappLoader 类的实例会在内部使用一个类载入器。回忆一下之前对 Loader 接口的讨论，该接口中声明了 getClassLoader() 方法，但是并没有声明 setClassLoader() 方法。因此，无法实例化一个新的类载入器，再将其赋给 WebappLoader 实例。难道说，WebappLoader 类的设计没有考虑到灵活性，只能使用默认类载入器吗？

答案是否定的。WebappLoader 类提供了 getLoaderClass() 方法和 setLoaderClass() 方法来获取或改变其私有变量 loaderClass 的值。该私有变量 loaderClass 保存了一个字符串类型的值，指明了类载入器的类的名字。默认情况下，变量 loaderClass 的值是 org.apache.catalina.loader.WebappClassLoader。也可以通过继承 WebappClassLoader 类的方式实现自己的类载入器，然后调用 setLoaderClass() 方法强制 WebappLoader 实例使用自定义类载入器。否则的话，在它启动时，WebappLoader 类会调用其私有方法 createClassLoader() 方法来创建默认类载入器。代码清单 8-3 给出了 createClassLoader() 方法的实现。

代码清单 8-3 createClassLoader 方法的实现

```
private WebappClassLoader createClassLoader() throws Exception {
    Class clazz = Class.forName(loaderClass);
    WebappClassLoader classLoader = null;
    if (parentClassLoader == null) {
        // Will cause a ClassCast if the class does not extend
        // WebappClassLoader, but this is on purpose (the exception will be
        // caught and rethrown)
        classLoader = (WebappClassLoader) clazz.newInstance();
        // in Tomcat 5, this if block is replaced by the following:
        // if (parentClassLoader == null) {
        //     parentClassLoader =
        //         Thread.currentThread().getContextClassLoader();
        // }
    }
    else {
        Class[] argTypes = { ClassLoader.class };
        Object[] args = { parentClassLoader };
        Constructor constr = clazz.getConstructor(argTypes);
        classLoader = (WebappClassLoader) constr.newInstance(args);
    }
    return classLoader;
}
```



可以不使用 `WebappClassLoader` 类的实例，而使用其他类的实例作为类载入器。但是请注意，`createClassLoader()` 方法的返回值是 `WebappClassLoader` 类型的。因此，如果自定义类载入器没有继承自 `WebappClassLoader` 类，`createClassLoader()` 方法就会抛出一个异常。

### 8.4.2 设置仓库

`WebappLoader` 类的 `start()` 方法会调用 `setRepositories()` 方法将仓库添加到其类载入器中。`WEB-INF/classes` 目录被传入到类载入器的 `addRepository()` 方法中，而 `WEB-INF/lib` 目录被传入到类载入器的 `setJarPath()` 方法中。这样，类载入器就能在 `WEB-INF/classes` 目录中和从部署的库到 `WEB-INF/lib` 目录载入相关类。

### 8.4.3 设置类路径

设置类路径的任务是通过在 `start()` 方法中调用 `setClassPath()` 方法完成的。`setClassPath()` 方法会在 `Servlet` 上下文中为 Jasper JSP 编译器设置一个字符串形式的属性来指明类路径信息。这里并不打算讨论与 JSP 编译相关的内容。

### 8.4.4 设置访问权限

若运行 Tomcat 时，使用了安全管理器，则 `setPermissions()` 方法会为类载入器设置访问相关目录的权限，例如，只能访问 `WEB-INF/classes` 和 `WEB-INF/lib` 和目录。若是没有使用安全管理器，则 `setPermissions()` 方法只是简单地返回，什么也不做。

### 8.4.5 开启新线程执行类的重新载入

`WebappLoader` 类支持自动重载功能。如果 `WEB-INF/classes` 目录或 `WEB-INF/lib` 目录下的某些类被重新编译了，那么这个类会自动重新载入，而无须重启 Tomcat。为了实现此目的，`WebappLoader` 类使用一个线程周期性地检查每个资源的时间戳。间隔时间由变量 `checkInterval` 指定，单位为秒。默认情况下，`checkInterval` 的值为 15，即每隔 15 秒会检查一次是否有文件需要自动重新载入。`getCheckInterval()` 方法和 `setCheckInterval()` 方法用于获取或设置间隔时间。

在 Tomcat 4 中，`WebappLoader` 类实现 `java.lang.Runnable` 接口来支持自动重载。代码清单 8-4 给出了 `WebappLoader` 类中 `run()` 方法的实现。

代码清单 8-4 `WebappLoader` 类中 `run()` 方法的实现

```
public void run() {
    if (debug >= 1)
        log("BACKGROUND THREAD Starting");

    // Loop until the termination semaphore is set
    while (!threadDone) {
        // Wait for our check interval
        threadSleep();
        if (!started)
            break;
        try {
            // Perform our modification check
```

```

        if (!classLoader.modified())
            continue;
    }
    catch (Exception e) {
        log(sm.getString("webappLoader.failModifiedCheck"), e);
        continue;
    }
    // Handle a need for reloading
    notifyContext();
    break;
}

if (debug >= 1)
    log("BACKGROUND THREAD Stopping");
}

```

**注意** 在 Tomcat 5 中, 检查类是否被修改的任务改为由 `org.apache.catalina.core.StandardContext` 类的 `backgroundProcess()` 方法完成。这个方法被 `org.apache.catalina.core.ContainerBase` 类中的一个专用线程周期性地调用, 该类是 `StandardContext` 类的父类。检查 `ContainerBase` 类的内部类 `ContainerBackgroundProcessor`, 后者实现了 `Runnable` 接口。

在代码清单 8-4 的 `run()` 方法中, 使用一个 `while` 循环来进行检查。循环体会一直执行下去, 直到变量 `started` 被设置为 `false` (变量 `started` 用来指明 `WebappLoader` 实例是否已经启动)。而 `while` 循环会执行以下操作:

- 使线程休眠一段时间, 时长由变量 `checkInterval` 指定, 单位为秒;
- 调用 `WebappLoader` 实例的类加载器的 `modified()` 方法检查已经载入的类是否被修改。若没有类修改, 则重新执行循环;
- 若某个已经载入的类被修改了, 则调用私有方法 `notifyContext()`, 通知与 `WebappLoader` 实例关联的 `Context` 容器重新载入相关类。

代码清单 8-5 给出了 `notifyContext()` 方法的实现。

代码清单 8-5 `notifyContext()` 方法的实现

```

private void notifyContext() {
    WebappContextNotifier notifier = new WebappContextNotifier();
    (new Thread(notifier)).start();
}

```

`notifyContext()` 方法并不会直接调用 `Context` 接口的 `reload()` 方法。相反, `notifyContext()` 方法会实例化一个内部类 `WebappContextNotifier`, 然后将其传入一个新建的线程对象, 并调用线程对象的 `start()` 方法。这样, 重载相关类的任务就可以在另一个线程中完成。代码清单 8-6 给出了 `WebappContextNotifier` 类的定义。

代码清单 8-6 `WebappContextNotifier` 类的定义

```

protected class WebappContextNotifier implements Runnable {
    public void run() {
        ((Context) container).reload();
    }
}

```

当将 `WebappContextNotifier` 实例传递给一个线程对象，并调用线程对象的 `start()` 方法时，会执行 `WebappContextNotifier` 实例的 `run()` 方法。然后，`run()` 方法会调用 `Context` 接口的 `reload()` 方法。第 12 章会在 `org.apache.catalina.core.StandardContext` 类中详细介绍 `reload()` 方法的实现方式。

## 8.5 WebappClassLoader 类

Web 应用程序中负责载入类的类载入器是 `org.apache.catalina.loader.WebappClassLoader` 类的实例。`WebappClassLoader` 类继承自 `java.net.URLClassLoader` 类，在前面的章节中，我们曾经使用 `URLClassLoader` 类来载入应用程序所需要用到的 Java 类。

`WebappClassLoader` 的设计方案考虑了优化和安全两方面。例如，它会缓存之前已经载入的类来提升性能。此外，它还会缓存加载失败的类的名字，这样，当再次请求加载同一个类时，类载入器就会直接抛出 `ClassNotFoundException` 异常，而不会再尝试查找该类了。`WebappClassLoader` 会在仓库列表和指定的 JAR 文件中搜索需要载入的类。

考虑到安全性，`WebappClassLoader` 类不允许载入指定的某些类，这些类的名字存储在一个字符串数组变量 `triggers` 中，当前只有一个元素：

```
private static final String[] triggers = {  
    "javax.servlet.Servlet"           // Servlet API  
};
```

此外，某些特殊的包及其子包下的类也是不允许载入的，也不会将载入类的任务委托给系统类载入器去执行：

```
private static final String[] packageTriggers = {  
    "javax",                          // Java extensions  
    "org.xml.sax",                     // SAX 1 & 2  
    "org.w3c.dom",                     // DOM 1 & 2  
    "org.apache.xerces",               // Xerces 1 & 2  
    "org.apache.xalan"                // Xalan  
};
```

下面几节说明 `WebappClassLoader` 类是如何完成待加载类的缓存和载入任务的。

### 8.5.1 类缓存

为了达到更好的性能，会缓存已经载入的类，这样，下次再使用该类时，会直接从缓存中获取。缓存可以在本地执行，即可以由 `WebappClassLoader` 实例来管理它所加载并缓存的类。此外，`java.lang.ClassLoader` 类会维护一个 `Vector` 对象，保存已经载入的类，防止这些类在不使用时当做垃圾而回收。在这种情况下，缓存是由父类来管理的。

每个由 `WebappClassLoader` 载入的类（无论是在 `WEB-INF/classes` 目录下还是从某个 JAR 文件内作为类文件部署），都视为“资源”。资源是 `org.apache.catalina.loader.ResourceEntry` 类的实例。`ResourceEntry` 实例会保存其所代表的 class 文件的字节流、最后一次修改日期、Manifest 信息（如果资源来自与一个 JAR 文件的话）等。

代码清单 8-7 给出了 `ResourceEntry` 类的定义。



代码清单 8-7 ResourceEntry 类的定义

```
package org.apache.catalina.loader;
import java.net.URL;
import java.security.cert.Certificate;
import java.util.jar.Manifest;

public class ResourceEntry {
    public long lastModified = -1;
    // Binary content of the resource.
    public byte[] binaryContent = null;
    public Class loadedClass = null;
    // URL source from where the object was loaded.
    public URL source = null;
    // URL of the codebase from where the object was loaded.
    public URL codeBase = null;
    public Manifest manifest = null;
    public Certificate[] certificates = null;
}
```

所有已经缓存的类会存储在一个名为 resourceEntries 的 HashMap 类型的变量中，其 key 值就是载入的资源名称。那些载入失败的类被存储到另一个名为 notFoundResources 的 HashMap 类型的变量中。

### 8.5.2 载入类

载入类时，WebappClassLoader 类要遵守如下规则：

- 因为所有已经载入的类都会缓存起来，所以载入类时要先检查本地缓存；
- 若本地缓存中没有，则检查上一层缓存，即调用 java.lang.ClassLoader 类的 findLoadedClass() 方法；
- 若两个缓存中都没有，则使用系统的类载入器进行加载，防止 Web 应用程序中的类覆盖 J2EE 的类；
- 若启用了 SecurityManager，则检查是否允许载入该类。若该类是禁止载入的类，抛出 ClassNotFoundException 异常；
- 若打开标志位 delegate，或者待载入的类是属于包触发器中的包名，则调用父载入器来载入相关类。如果父类载入器为 null，则使用系统的类载入器；
- 从当前仓库中载入相关类；
- 若当前仓库中没有需要的类，且标志位 delegate 关闭，则使用父类载入器。若父类载入器为 null，则使用系统的类载入器进行加载；
- 若仍未找到需要的类，则抛出 ClassNotFoundException 异常。

### 8.5.3 应用程序

本章的程序用于说明如何使用与某个 Context 容器相关联的 WebappLoader 实例。在 Tomcat 中，Context 接口的标准实现是 org.apache.catalina.core.StandardContext，因此，本章的应用程序会使用 StandardContext 实例作为 servlet 容器。但是，对 StandardContext 类的详细讨论将会

在第12章中进行。现在你还不需要了解 StandardContext 类的太多实现细节。你只需要知道 StandardContext 类是如何与监听它触发的事件（如 START\_EVENT 事件和 STOP\_EVENT 事件）的监听器协同工作的。监听器必须实现 org.apache.catalina.lifecycle.LifecycleListener 接口，并调用 StandardContext 类的 setConfigured() 方法。对于本章的应用程序，监听器是 ex08.pyrmont.core.SimpleContextConfig 类的实例。代码清单 8-8 给出了 SimpleContextConfig 类的定义。

代码清单 8-8 SimpleContextConfig 类的定义

```
package ex08.pyrmont.core;
import org.apache.catalina.Context;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleEvent;
import org.apache.catalina.LifecycleListener;

public class SimpleContextConfig implements LifecycleListener {
    public void lifecycleEvent(LifecycleEvent event) {
        if (Lifecycle.START_EVENT.equals(event.getType())) {
            Context context = (Context) event.getLifecycle();
            context.setConfigured(true);
        }
    }
}
```

需要的做是实例化 StandardContext 类和 SimpleContextConfig 类，然后，调用 org.apache.catalina.Lifecycle 接口的 addLifecycleListener() 方法，通过 StandardContext 注册 SimpleContextConfig。Lifecycle 接口的详细内容请参见第6章。

此外，该应用程序保留了第7章中应用程序中的一些类，如 SimplePipeline、SimpleWrapper 和 SimpleWrapperValve。

本章的应用程序可以使用 PrimitiveServlet 类和 ModernServlet 类进行测试，但是，这里使用了 StandardContext 类的实例作为 servlet 容器，所以 servlet 类只能放置于应用程序目录的 WEB-INF/classes 目录下。应用程序目录名为 myApp，在第1次部署可下载的 ZIP 文件时，应该已经创建好了。为了通知 StandardContext 实例到哪里查找应用程序目录，需要设置一个名为“catalina.base”的系统属性，其值为“user.dir”属性的值，如下所示：

```
System.setProperty("catalina.base", System.getProperty("user.dir"));
```

事实上，上面的代码就是 Bootstrap 类的 main() 方法中的第1行内容。然后，main() 方法会实例化默认连接器：

```
Connector connector = new HttpConnector();
```

然后，与之前章节中的应用程序一样，它为两个 servlet 类创建两个 Wrapper 实例：

```
Wrapper wrapper1 = new SimpleWrapper();
wrapper1.setName("Primitive");
wrapper1.setServletClass("PrimitiveServlet");
Wrapper wrapper2 = new SimpleWrapper();
wrapper2.setName("Modern");
wrapper2.setServletClass("ModernServlet");
```

接着，它再创建一个 `StandardContext` 实例，设置应用程序路径和上下文的文档根路径：

```
Context context = new StandardContext();
// StandardContext's start method adds a default mapper
context.setPath("/myApp");
context.setDocBase("myApp");
```

上面的代码在功能上等同于下面在 `server.xml` 文件中的配置：

```
<Context path="/myApp" docBase="myApp"/>
```

然后，将两个 `Wrapper` 实例添加到 `Context` 容器中，为它们设置访问路径的映射关系，这样，`Context` 容器就能够定位到它们：

```
context.addChild(wrapper1);
context.addChild(wrapper2);
context.addServletMapping("/Primitive", "Primitive");
context.addServletMapping("/Modern", "Modern");
```

下一步，实例化一个监听器，并通过 `Context` 容器注册它：

```
LifecycleListener listener = new SimpleContextConfig();
((Lifecycle) context).addLifecycleListener(listener);
```

接着，它会实例化 `WebappLoader` 类，并将其关联到 `Context` 容器：

```
Loader loader = new WebappLoader();
context.setLoader(loader);
```

然后，将 `Context` 容器与默认连接器相关联，调用默认连接器的 `initialize()` 和 `start()` 方法，再调用 `Context` 容器的 `start()` 方法，这样 `Servlet` 容器准备就绪，可以处理 `Servlet` 请求了：

```
connector.setContainer(context);
try {
    connector.initialize();
    ((Lifecycle) connector).start();
    ((Lifecycle) context).start();
}
```

接下来的几行代码仅仅显示出资源的 `docBase` 属性值和类载入器中所有的仓库的名字：

```
// now we want to know some details about WebappLoader
WebappClassLoader classLoader = (WebappClassLoader)
    loader.getClassLoader();
System.out.println("Resources' docBase: " +
    ((ProxyDirContext) classLoader.getResources()).getDocBase());
String[] repositories = classLoader.findRepositories();
for (int i=0; i<repositories.length; i++) {
    System.out.println(" repository: " + repositories[i]);
}
```

下面的几行内容就是运行应用程序后，显示出的 `docBase` 属性值和所有的仓库的名字：

```
Resources' docBase: C:\HowTomcatWorks\myApp
repository: /WEB-INF/classes/
```

`docBase` 属性的值会根据应用程序安装在哪里而有所变化。

最后，应用程序在用户在控制台上按 `Enter` 键后退出：

```
// make the application wait until we press a key.
```



```
System.in.read();  
((Lifecycle) context).stop();
```

## 8.6 运行应用程序

要在 Windows 平台下运行该应用程序，需要在工作目录下执行如下命令：

```
java -classpath ./lib/servlet.jar;./lib/commons-collections.jar;./ex08.pyrmont.startup.Bootstrap
```

要在 Linux 平台下运行，需要使用冒号来代替库文件之间的分号：

```
java -classpath ./lib/servlet.jar:./lib/commons-collections.jar:./ex08.pyrmont.startup.Bootstrap
```

要想调用 `PrimitiveServlet`，可以使用如下的 URL：

```
http://localhost:8080/Primitive
```

类似地，要调用 `ModernServlet` 时，可以使用如下的 URL：

```
http://localhost:8080/Modern
```

## 8.7 小结

Web 应用程序中的载入器，或一个简单的载入器，都是 Catalina 中最重要的组件。载入器负责载入应用程序所需要的类，因此会使用一个内部类载入器。这个内部类载入器是一个自定义类，Tomcat 使用这个自定义的类载入器对 Web 应用程序上下文中要载入的类进行一些约束。此外，自定义类载入器可以支持对载入类的缓存和对一个或多个被修改的类的自动重载。

## 第 9 章

# Session 管理

Catalina 通过一个称为 Session 管理器的组件来管理建立的 Session 对象，该组件由 `org.apache.catalina.Manager` 接口表示。Session 管理器需要与一个 Context 容器相关联，且必须与一个 Context 容器关联。相比于其他组件，Session 管理器负责创建、更新、销毁 Session 对象，当有请求到来时，要返回一个有效的 Session 对象。

servlet 实例可以通过调用 `javax.servlet.http.HttpServletRequest` 接口的 `getSession()` 方法来获取一个 Session 对象。在 Catalina 的默认连接器中，`org.apache.catalina.connector.HttpRequestBase` 类实现 `HttpServletRequest` 接口，可以用来获取 Session 对象。下面是 `HttpRequestBase` 类中的一些相关方法。

```
public HttpSession getSession() {
    return (getSession(true));
}

public HttpSession getSession(boolean create) {
    ...
    return doGetSession(create);
}

private HttpSession doGetSession(boolean create) {
    // There cannot be a session if no context has been assigned yet
    if (context == null)
        return (null);
    // Return the current session if it exists and is valid
    if ((session != null) && !session.isValid())
        session = null;
    if (session != null)
        return (session.getSession());

    // Return the requested session if it exists and is valid
    Manager manager = null;
    if (context != null)
        manager = context.getManager();
    if (manager == null)
        return (null);    // Sessions are not supported
    if (requestedSessionId != null) {
        try {
            session = manager.findSession(requestedSessionId);
        }
        catch (IOException e) {
            session = null;
        }
    }
    if ((session != null) && !session.isValid())
        session = null;
    if (session != null) {
        return (session.getSession());
    }
}
```

```

// Create a new session if requested and the response is not
// committed
if (!create)
    return (null);
...
session = manager.createSession();
if (session != null)
    return (session.getSession());
else
    return (null);
}

```

默认情况下，Session 管理器会将其所管理的 Session 对象存放在内存中。但是，在 Tomcat 中，Session 管理器也可以将 Session 对象进行持久化，存储到文件存储器或通过 JDBC 写入到数据库中。在 Catalina 中，org.apache.catalina.session 包下有一些与 Session 对象和 Session 对象管理相关的类。

9.1 ~ 9.3 节说明 Catalina 是如何管理 Session 对象的，9.4 节说明应用程序如何通过关联管理器来使用 Context 容器。

## 9.1 Session 对象

在 servlet 编程方面中，Session 对象由 javax.servlet.http.HttpSession 接口表示。在 Catalina 中 Session 接口的标准实现是位于 org.apache.catalina.session 包下的 StandardSession 类。但是，为了安全起见，Session 管理器并不会直接将 StandardSession 实例交给 servlet 实例使用。相反，这里使用了一个 Session 接口的外观类，StandardSessionFacade，它位于 org.apache.catalina.session 包下。在内部，Session 管理器会使用另一个外观类：org.apache.catalina.Session 接口。图 9-1 展示了与 Session 管理相关的一些类的 UML 类图。需要注意的是，Session、StandardSession 和 StandardSessionFacade 类名中的 org.apache.catalina 前缀省略掉了。

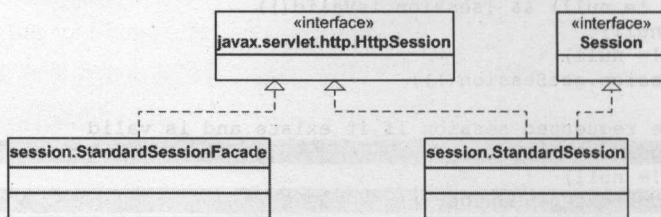


图 9-1 Session 管理相关类的 UML 类图

### 9.1.1 Session 接口

Session 接口是作为 Catalina 内部的外观类使用的。Session 接口的标准实现 StandardSession 类也实现了 javax.servlet.http.HttpSession 接口。代码清单 9-1 给出了 Session 接口的定义。

代码清单 9-1 Session 接口的定义

```

package org.apache.catalina;
import java.io.IOException;
import java.security.Principal;

```



```
import java.util.Iterator;
import javax.servlet.ServletException;
import javax.servlet.http.HttpSession;

public interface Session {
    public static final String SESSION_CREATED_EVENT = "createSession";
    public static final String SESSION_DESTROYED_EVENT = "destroySession";
    public String getAuthType();
    public void setAuthType(String authType);
    public long getCreationTime();
    public void setCreationTime(long time);
    public String getId();
    public void setId(String id);
    public String getInfo();
    public long getLastAccessedTime();
    public Manager getManager();
    public void setManager(Manager manager);
    public int getMaxInactiveInterval();
    public void setMaxInactiveInterval(int interval);
    public void setNew(boolean isNew);
    public Principal getPrincipal();
    public void setPrincipal(Principal principal);
    public HttpSession getSession();
    public void setValid(boolean isValid);
    public boolean isValid();
    public void access();
    public void addSessionListener(SessionListener listener);
    public void expire();
    public Object getNote(String name);
    public Iterator getNoteNames();
    public void recycle();
    public void removeNote(String name);
    public void removeSessionListener(SessionListener listener);
    public void setNote(String name, Object value);
}
```

Session 对象总是存在于 Session 管理器中的，可以使用 `getManager()` 方法或 `setManager()` 方法将 Session 实例和某个 Session 管理器相关联。对某个 Session 实例来说，在与其 Session 管理器相关联的某个 Context 实例内，该 Session 对象有一个唯一的标识符，可以通过 `setId()` 方法和 `getId()` 方法来访问该 Session 标识符。Session 管理器会调用 `getLastAccessedTime()` 方法，根据返回值来判断一个 Session 对象的有效性。Session 管理器会调用 `setValid()` 方法来设置或重置 Session 对象的有效性。每当访问一个 Session 实例时，会调用其 `access()` 方法来修改 Session 对象的最后访问时间。最后，Session 管理器会调用 Session 对象的 `expire()` 方法将其设置为过期，也可以通过 `getSession()` 方法获取一个经过 Session 外观类包装的 `HttpSession` 对象。

### 9.1.2 StandardSession 类

StandardSession 类是 Session 接口的标准实现。除了实现 `javax.servlet.http.HttpSession` 接口和 `org.apache.catalina.Session` 接口外，StandardSession 类还实现 `java.lang.Serializable` 接口，便于序列化 Session 对象。

StandardSession 类的构造函数接受 Manager 接口的一个实例，迫使一个 Session 对象必须拥

有一个 Session 管理器实例:

```
public StandardSession(Manager manager);
```

下面是 StandardSession 实例的一些比较重要的私有变量, 用于维护该 StandardSession 实例的一些状态。注意, 带有 transient 关键字的变量无法序列化。

```
// session attributes
private HashMap attributes = new HashMap();
// the authentication type used to authenticate our cached Principal,
if any
private transient String authType = null;
private long creationTime = 0L;
private transient boolean expiring = false;
private transient StandardSessionFacade facade = null;
private String id = null;
private long lastAccessedTime = creationTime;
// The session event listeners for this Session.
private transient ArrayList listeners = new ArrayList();
private Manager manager = null;
private int maxInactiveInterval = -1;
// Flag indicating whether this session is new or not.
private boolean isNew = false;
private boolean isValid = false;
private long thisAccessedTime = creationTime;
```

**注意** 在 Tomcat 5 中, 上述变量是受保护的, 而 Tomcat 4 中, 它们都是私有变量。每个变量都有与之对应的一个访问器和一个转变器 (get 和 set 方法)。

其中 getSession() 方法会通过传入一个自身实例来创建 StandardSessionFacade 类的一个实例, 并将其返回:

```
public HttpSession getSession() {
    if (facade == null)
        facade = new StandardSessionFacade(this);
    return (facade);
}
```

若 Session 管理器中的某个 Session 对象在某个时间长度内都没有被访问的话, 会被 Session 管理器设置为过期, 这个时间长度由变量 maxInactiveInterval 的值来指定。将一个 Session 对象设置为过期是通过调用 Session 接口的 expire() 方法完成的。代码清单 9-2 给出了 Tomcat 4 中 StandardSession 类的 expire() 方法的实现。

代码清单 9-2 expire() 方法的实现

```
public void expire(boolean notify) {
    // Mark this session as "being expired" if needed
    if (expiring)
        return;
    expiring = true;
    setValid(false);

    // Remove this session from our manager's active sessions
    if (manager != null)
        manager.remove(this);
    // Unbind any objects associated with this session
```

```

String keys[] = keys();
for (int i = 0; i < keys.length; i++)
    removeAttribute(keys[i], notify);
// Notify interested session event listeners
if (notify) {
    fireSessionEvent(Session.SESSION_DESTROYED_EVENT, null);
}
// Notify interested application event listeners
// FIXME - Assumes we call listeners in reverse order
Context context = (Context) manager.getContainer();
Object listeners[] = context.getApplicationListeners();
if (notify && (listeners != null)) {
    HttpSessionEvent event = new HttpSessionEvent(getSession());
    for (int i = 0; i < listeners.length; i++) {
        int j = (listeners.length - 1) - i;
        if (!(listeners[j] instanceof HttpSessionListener))
            continue;
        HttpSessionListener listener =
            (HttpSessionListener) listeners[j];
        try {
            fireContainerEvent(context, "beforeSessionDestroyed",
                listener);
            listener.sessionDestroyed(event);
            fireContainerEvent(context, "afterSessionDestroyed", listener);
        }
        catch (Throwable t) {
            try {
                fireContainerEvent(context, "afterSessionDestroyed",
                    listener);
            }
            catch (Exception e) {
                ;
            }
        }
        // FIXME - should we do anything besides log these?
        log(sm.getString("standardSession.sessionEvent"), t);
    }
}
}

// We have completed expire of this session
expiring = false;
if ((manager != null) && (manager instanceof ManagerBase)) {
    recycle();
}
}

```

代码清单 9-2 展示的 `expire()` 方法中的操作包括设置一个名为 `expiring` 的内部变量，从 Session 管理器中移除 Session 实例和触发一些事件。

### 9.1.3 StandardSessionFacade 类

为了传递一个 Session 对象给 servlet 实例，Catalina 会实例化 `StandardSession` 类，填充该 Session 对象，然后再将其传给 servlet 实例。但是，实际上，Catalina 传递的是 Session 的外观类 `StandardSessionFacade` 的实例，该类仅仅实现了 `javax.servlet.http.HttpSession` 接口中的方法。这样，servlet 程序员就不能将 `HttpSession` 对象向下转换为 `StandardSession` 类型，也阻止了 servlet



程序员访问一些敏感方法。

## 9.2 Manager

Session 管理器组件负责管理 Session 对象，例如，创建和销毁 Session 对象。Session 管理器是 org.apache.catalina.Manager 接口的实例。在 Catalina 中，org.apache.catalina.session 包中有一个名为 ManagerBase 的工具类，该类提供了常见功能的实现。ManagerBase 类有两个直接子类，分别是 StandardManager 类和 PersistentManagerBase 类。

当 Catalina 运行时，StandardManager 实例会将 Session 对象存储在内存中。但是，当 Catalina 关闭时，它会将当前内存中的所有 Session 对象存储到一个文件中。当再次启动 Catalina 时，又会将这些 Session 对象重新载入到内存中。

有一类 Session 管理器会将 Session 对象存储到辅助存储器中，PersistentManagerBase 类就是这类 Session 管理器的基类。PersistentManagerBase 类有两个子类：PersistentManager 类和 DistributedManager 类（该类只存在于 Tomcat 4 中）。图 9-2 给出了 Manager 接口及其实现类的 UML 类图。

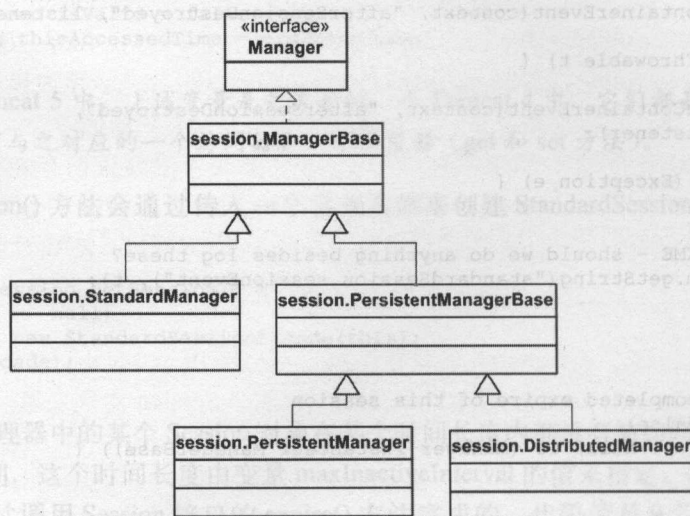


图 9-2 Manager 接口及其相关类的 UML 类图

### 9.2.1 Manager 接口

Session 管理器是 Manager 接口的实例。代码清单 9-3 给出了 Manager 接口的定义。

代码清单 9-3 Manager 接口的定义

```

package org.apache.catalina;
import java.beans.PropertyChangeListener;
import java.io.IOException;

public interface Manager {
    public Container getContainer();
  
```

```
public void setContainer(Container container);
public DefaultContext getDefaultContext();
public void setDefaultContext(DefaultContext defaultContext);
public boolean getDistributable();
public void setDistributable(boolean distributable);
public String getInfo();
public int getMaxInactiveInterval();
public void setMaxInactiveInterval(int interval);
public void add(Session session);
public void addPropertyChangeListener(PropertyChangeListener
    listener);
public Session createSession();
public Session findSession(String id) throws IOException;
public Session[] findSessions();
public void load() throws ClassNotFoundException, IOException;
public void remove(Session session);
public void removePropertyChangeListener(PropertyChangeListener
    listener);
public void unload() throws IOException;
}
```

首先, Manager 接口提供了 `getContainer()` 方法和 `setContainer()` 方法, 以便将一个 Manager 实现与一个 Context 容器相关联。 `createSession()` 方法用来创建一个 Session 实例, `add()` 方法会将一个 Session 实例添加到 Session 池中, 而 `remove()` 方法则会将一个 Session 实例从 Session 池中移除。 `getMaxInactiveInterval()` 方法和 `setMaxInactiveInterval()` 方法用来获取或设置一个时间长度, 单位为秒。 Session 管理器会以此值作为一个 Session 对象的最长存活时间。

最后, `load()` 方法和 `unload()` 方法用来将 Session 对象持久化到辅助存储器中, 当然这还需要 Session 管理器对持久化机制的支持。 `unload()` 方法会将当前活动的 Session 对象存储到 Manager 实现指定的介质中, 而 `load()` 方法会将介质中的 Session 对象重新载入到内存中。

## 9.2.2 ManagerBase 类

ManagerBase 类是一个抽象类, 所有的 Session 管理器组件都会继承此类。 该类提供了很多功能方便其子类使用。 此外, ManagerBase 类的 `createSession()` 方法会创建一个新的 Session 对象。 每个 Session 对象都有一个唯一的标识符, 可以通过 ManagerBase 类受保护的方法 `generateSessionId()` 方法来返回一个唯一的标识符。

**注意** 一个活动的 Session 对象指的是有效的、还未过期的 Session 对象。

某个 Context 容器的 Session 管理器会管理该 Context 容器中所有活动的 Session 对象。 这些活动的 Session 对象都存储在一个名为 `sessions` 的 HashMap 变量中:

```
protected HashMap sessions = new HashMap();
```

`add()` 方法会将一个 Session 对象添加到 HashMap 变量 `sessions` 中, 如下所示:

```
public void add(Session session) {
    synchronized (sessions) {
        sessions.put(session.getId(), session);
    }
}
```

`remove()` 方法会将一个 `Session` 对象从 `HashMap` 变量 `sessions` 中移除，如下所示：

```
public void remove(Session session) {
    synchronized (sessions) {
        sessions.remove(session.getId());
    }
}
```

不带参数的 `findSession()` 方法会以 `Session` 实例数组的形式从 `HashMap` 变量 `sessions` 中返回所有活动的 `Session` 对象，而带参数的 `findSession()` 方法接收一个 `Session` 对象的标识符作为参数，并将该标识符指定的 `Session` 对象返回。这两个方法的定义如下所示：

```
public Session[] findSessions() {
    Session results[] = null;
    synchronized (sessions) {
        results = new Session[sessions.size()];
        results = (Session[]) sessions.values().toArray(results);
    }
    return (results);
}

public Session findSession(String id) throws IOException {
    if (id == null)
        return (null);
    synchronized (sessions) {
        Session session = (Session) sessions.get(id);
        return (session);
    }
}
```

### 9.2.3 StandardManager 类

`StandardManager` 类是 `Manager` 接口的标准实现，该类将 `Session` 对存储于内存中。`StandardManager` 类实现 `Lifecycle` 接口（详细内容参见第6章），这样就可以由与其关联的 `Context` 容器来启动或关闭。其中 `stop()` 方法的实现会调用 `unload()` 方法，以便将有效的 `Session` 对象序列化到一个名为“`SESSION.ser`”的文件中，而且每个 `Context` 容器都会产生一个这样的文件。`SESSION.ser` 文件位于环境变量 `CATALINA_HOME` 指定的目录下的 `work` 目录中。例如，在 Tomcat 4 和 Tomcat 5 中，如果运行了示例应用程序，就可以在 `CATALINA_HOME/work/Standalone/localhost/examples` 目录下找到 `SESSION.ser` 文件。当 `StandardManager` 实例再次启动时，这些 `Session` 对象会通过调用 `load()` 方法重新读入内存中。

`Session` 管理器还要负责销毁那些已经失效的 `Session` 对象。在 Tomcat 4 的 `StandardManager` 类中，这项工作是由一个专门的线程来完成的。为此，`StandardManager` 类还实现了 `java.lang.Runnable` 接口。代码清单 9-4 展示了 `StandardManager` 类中 `run()` 方法的定义。

代码清单 9-4 `StandardManager` 类中 `run()` 方法的定义

```
public void run() {
    // Loop until the termination semaphore is set
    while (!threadDone) {
        threadSleep();
        processExpires();
    }
}
```



threadSleep() 方法会使线程休眠一段时间（单位为秒），时间长度由变量 checkInterval 指定，默认情况下为 60 秒。可以调用 setCheckInterval() 方法来修改这个值。

processExpire() 方法会遍历由 Session 管理器管理的所有 Session 对象，将 Session 实例的 lastAccessedTime 属性值与当前时间进行比较。如果两者之间的差值超过了变量 maxInactiveInterval 指定的数值，则会调用 Session 接口的 expire() 方法使这个 Session 实例过期。变量 maxInactiveInterval 的值可以通过调用 setMaxInactiveInterval() 方法进行设置，在 StandardManager 类中，该值默认为 60。但是，不要天真地以为在实际部署 Tomcat 时用的还是这个值。org.apache.catalina.core.ContainerBase 类的 setManager() 方法（总是需要调用 setManager() 方法将 Session 管理器与某个 Context 容器相关联）会调用 setContainer() 方法来重写此值。下面是 setContainer() 方法的代码片段：

```
setMaxInactiveInterval(((Context)
this.container).getSessionTimeout()*60);
```

**注意** 在 org.apache.catalina.core.StandardContext 类中，变量 sessionTimeout 的值默认为 30。

在 Tomcat 5 中，StandardManager 类已不再实现 java.lang Runnable 接口。相反，backgroundProcess() 方法会直接调用 Tomcat 5 中 StandardManager 对象的 processExpires() 方法。backgroundProcess() 方法并不存在于 Tomcat 4 中：

```
public void backgroundProcess() {
    processExpires();
}
```

org.apache.catalina.core.StandardContext 实例的 backgroundProcess() 方法会调用与其相关联的 StandardManager 实例的 backgroundProcess() 方法。而 StandardContext 实例会周期性的调用其 backgroundProcess() 方法（详细内容参见第 12 章）。

#### 9.2.4 PersistentManagerBase 类

PersistentManagerBase 类是所有持久化 Session 管理器的父类。StandardManager 实例和持久化 Session 管理器的区别在于后者中存储器的表现形式，即存储 Session 对象的辅助存储器的形式，使用 Store 对象来表示。PersistentManagerBase 类使用一个名为 store 的私有对象引用：

```
private Store store = null;
```

在持久化 Session 管理器中，Session 对象可以备份，也可以换出。当备份一个 Session 对象时，该 Session 对象会被复制到存储器中，而原对象仍留在内存中。因此，如果服务器崩溃，就可以从存储器中获取活动的 Session 对象。当 Session 对象被换出时，它会被移动到存储器中，因为当前活动的 Session 对象数超过了上限值，或者这个 Session 对象闲置了过长的时间。之所以换出是为了节省内存空间。

在 Tomcat 4 中，PersistentManagerBase 类实现 java.lang Runnable 接口，使用一个专门的线程来执行备份和换出活动的 Session 对象的任务。下面是 run() 方法的实现：



## 2. 备份

并非所有的 Session 对象都会备份。PersistentManagerBase 实例仅仅会备份那些空闲时间超过了变量 maxIdleBackup 指定的值的 Session 对象。processMaxIdleBackups() 方法负责对该 Session 对象进行备份操作。

### 9.2.5 PersistentManager 类

PersistentManager 类继承自 PersistentManagerBase 类，并没有添加其他的方法，只是多了两个属性。代码清单 9-5 给出了 PersistentManager 类的定义。

代码清单 9-5 PersistentManager 类的定义

```
package org.apache.catalina.session;

public final class PersistentManager extends PersistentManagerBase {
    // The descriptive information about this implementation.
    private static final String info = "PersistentManager/1.0";
    // The descriptive name of this Manager implementation (for logging).
    protected static String name = "PersistentManager";
    public String getInfo() {
        return (this.info);
    }
    public String getName() {
        return (name);
    }
}
```

### 9.2.6 DistributedManager 类

Tomcat 4 提供了 DistributedManager 类，该类继承自 PersistentManagerBase 类，前一个类用于两个或多个节点的集群环境。一个节点表示部署的一台 Tomcat 服务器。集群中的节点可以在同一台机器上，也可以在不同的机器上。在集群环境中，每个节点必须使用 DistributedManager 实例作为其 Session 管理器，这样才能支持复制 Session 对象，这也是 DistributedManager 类的主要功能。

为了实现复制 Session 对象的目的，当创建或销毁 Session 对象时，DistributedManager 实例会向其他节点发送消息。此外，集群中的节点也必须能够接收其他节点发送的消息。这样，HTTP 请求才能到达集群中的任意节点。

为了与集群中其他节点的 DistributedManager 实例发送和接收消息，Catalina 在 org.apache.catalina.cluster 包内有一些可供使用的工具类。其中，ClusterSender 类用于向集群中的其他节点发送消息，ClusterReceiver 类则用于接收集群中其他节点发送的消息。

DistributedManager 实例的 createSession() 方法要创建一个 Session 对象，存储在当前 DistributedManager 实例中，并使用 ClusterSender 实例向其他节点发送消息。代码清单 9-6 给出了 createSession() 方法的实现。



代码清单 9-6 createSession() 方法的实现

```

public Session createSession() {
    Session session = super.createSession();
    ObjectOutputStream oos = null;
    ByteArrayOutputStream bos = null;
    ByteArrayInputStream bis = null;

    try {
        bos = new ByteArrayOutputStream();
        oos = new ObjectOutputStream(new BufferedOutputStream(bos));
        ((StandardSession) session).writeObjectData(oos);
        oos.close();
        byte[] obs = bos.toByteArray();
        clusterSender.send(obs);
        if(debug > 0)
            log("Replicating Session: "+session.getId());
    }
    catch (IOException e) {
        log("An error occurred when replicating Session: " +
            session.getId());
    }
    return (session);
}

```

首先, createSession() 方法会调用超类的 createSession() 方法来为自身创建一个 Session 对象。然后, 它使用 ClusterSender 实例, 以字节数组的形式将该 Session 对象发送到集群中的其他节点。

DistribubedManager 类还实现了 java.lang Runnable 接口, 这样就可以使用一个专门的线程来检查 Session 对象是否过期, 并从集群中的其他节点上接收消息。run() 方法的实现如下所示:

```

public void run() {
    // Loop until the termination semaphore is set
    while (!threadDone) {
        threadSleep();
        processClusterReceiver();
        processExpires();
        processPersistenceChecks();
    }
}

```

需要注意的是, 调用处理 Session 的 processClusterReceiver() 方法来创建来自集群中其他节点的消息。

### 9.3 存储器

存储器是 org.apache.catalina.Store 接口的实例, 是为 Session 管理器管理的 Session 对象提供持久化存储器的一个组件。代码清单 9-7 给出了 Store 接口的定义。

代码清单 9-7 Store 接口的定义

```

package org.apache.catalina;
import java.beans.PropertyChangeListener;
import java.io.IOException;

```

```

public interface Store {
    public String getInfo();
    public Manager getManager();
    public void setManager(Manager manager);
    public int getSize() throws IOException;
    public void addPropertyChangeListener(PropertyChangeListener
        listener);
    public String[] keys() throws IOException;
    public Session load(String id)
        throws ClassNotFoundException, IOException;
    public void remove(String id) throws IOException;
    public void clear() throws IOException;
    public void removePropertyChangeListener(PropertyChangeListener
        listener);
    public void save(Session session) throws IOException;
}

```

Store 接口中比较重要的两个方法是 save() 方法和 load() 方法。save() 方法用于将指定的 Session 对象存储到某种持久性存储器中。而 load() 方法会从存储器中，依据 Session 对象的标识符将该 Session 对象载入到内存中。另外，keys() 方法会以字符串数组的形式返回所有 Session 对象的标识符。

图 9-3 展示了 Store 接口及其实现类的 UML 类图。注意，其中各类类名中的 org.apache.catalina 前缀省略掉了。

下面几节来对 StoreBase、FileStore 和 JDBCStore 类进行说明。

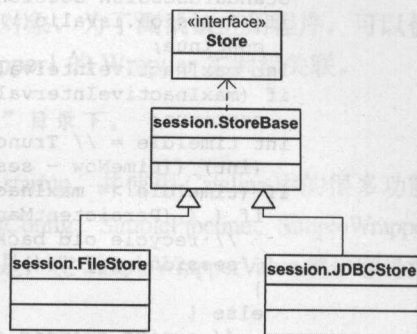


图 9-3 Store 接口及实现类的 UML 类图

### 9.3.1 StoreBase 类

StoreBase 类是一个抽象类，提供了一些基本功能。该类有两个直接子类，分别是 FileStore 类和 JDBCStore 类。StoreBase 类并没有实现 Store 接口的 save() 方法和 load() 方法，因为，这两个方法依赖于持久化 Session 对象的存储器的类型。

在 Tomcat 4 中，StoreBase 类使用另一个线程周期性地检查 Session 对象，从活动的 Session 的集合中移除过期的 Session 对象。下面是 Tomcat 4 中 StoreBase 类的 run() 方法的实现：

```

public void run() {
    // Loop until the termination semaphore is set
    while (!threadDone) {
        threadSleep();
        processExpires();
    }
}

```

processExpires() 方法会获取所有活动的 Session 对象，检查每个 Session 对象的 lastAccessedTime 属性值，删除那些长时间不活动的 Session 对象。代码清单 9-8 给出了 processExpires() 方法的实现。

代码清单 9-8 processExpires() 方法的实现

```

protected void processExpires() {
    long timeNow = System.currentTimeMillis();
    String[] keys = null;
    if (!started)
        return;
    try {
        keys = keys();
    }
    catch (IOException e) {
        log (e.toString());
        e.printStackTrace();
        return;
    }

    for (int i = 0; i < keys.length; i++) {
        try {
            StandardSession session = (StandardSession) load(keys[i]);
            if (!session.isValid())
                continue;
            int maxInactiveInterval = session.getMaxInactiveInterval();
            if (maxInactiveInterval < 0)
                continue;
            int timeIdle = // Truncate, do not round up
                (int) ((timeNow - session.getLastAccessedTime()) / 1000L);
            if (timeIdle >= maxInactiveInterval) {
                if ( ( (PersistentManagerBase) manager).isLoaded( keys[i] ) ) {
                    // recycle old backup session
                    session.recycle();
                }
                else {
                    // expire swapped out session
                    session.expire();
                }
                remove(session.getId());
            }
        }
        catch (IOException e) {
            log (e.toString());
            e.printStackTrace();
        }
        catch (ClassNotFoundException e) {
            log (e.toString());
            e.printStackTrace();
        }
    }
}

```

在 Tomcat 5 中，不再使用专用的线程调用 processExpires() 方法。相反，相关联的 PersistentManagerBase 实例的 backgroundProcess() 方法会周期性地调用 processExpires() 方法。

### 9.3.2 FileStore 类

FileStore 类会将 Session 对象存储到某个文件中。文件名会使用 Session 对象的标识符再加上一个后缀“.session”构成。文件位于临时的工作目录下，可以调用 FileStore 类的 setDirectory() 方法修改临时目录的位置。



save() 方法使用 java.io.ObjectOutputStream 类将 Session 对象进行序列化。因此, 所有存储在 Session 实例中的对象都要实现 java.lang.Serializable 接口。load() 方法使用 java.io.ObjectInputStream 类实现 Session 对象的反序列化。

### 9.3.3 JDBCStore 类

JDBCStore 类将 Session 对象通过 JDBC 存入数据库中。因此, 为了使用 JDBCStore, 需要分别调用 setDriverName() 方法和 setConnectionURL() 方法来设置驱动程序名称和连接 URL。

## 9.4 应用程序

本章的应用程序与第 8 章中的类似, 它使用默认的连接器和一个 Context 实例作为其主 servlet 容器, 并配备了一个 Wrapper 实例。但是, 其中的一个区别是, 本章应用程序中的 Context 容器使用一个 StandardManager 实例来管理 Session 对象。为了测试该应用程序, 可以使用第 3 个示例 servlet: SessionServlet。该 servlet 与名为 wrapper1 的 Wrapper 实例相关联。

**注意** SessionServlet 类位于 “myApp/WEB-INF/classes” 目录下。

该应用程序有两个包: ex09.pyrmont.core 和 ex09.pyrmont.startup, 并使用 Catalina 中的很多功能类。在 ex09.pyrmont.core 包下有 4 个类, 分别是 SimpleContextConfig、SimplePipeline、SimpleWrapper 和 SimpleWrapperValve 类。前 3 个类与第 8 章中的类相同, 但是, 在 SimpleWrapperValve 类中额外添加了两行代码。ex09.pyrmont.startup 包下有 1 个类: Bootstrap。

Bootstrap 类会在下面的 9.4.1 节中讨论, SimpleWrapperValve 类在 9.4.2 节中讨论。9.4.3 节会说明如何启动应用程序。

### 9.4.1 Bootstrap 类

Bootstrap 类用来启动应用程序, 与第 8 章中的 Bootstrap 类很相似。但是, 本章中的 Bootstrap 类会创建 org.apache.catalina.session.StandardManager 类的一个实例, 并将其与一个 Context 实例相关联。

main() 方法中会先设置系统属性 “catalina.base” 的值, 并实例化默认连接器。

```
System.setProperty("catalina.base",  
    System.getProperty("user.dir"));  
Connector connector = new HttpConnector();
```

对于 servlet 类 SessionServlet, 它会创建一个名为 wrapper1 的 Wrapper 实例:

```
Wrapper wrapper1 = new SimpleWrapper();  
wrapper1.setName("Session");  
wrapper1.setServletClass("SessionServlet");
```

然后, 它创建一个 StandardContext 对象, 设置其 path 和 docBase 属性, 并将 Wrapper 实例添加到 Context 容器中:

```
Context context = new StandardContext();  
context.setPath("/myApp");  
context.setDocBase("myApp");  
context.addChild(wrapper1);
```

接下来, `start()` 方法添加 `Servlet` 映射。这里的映射与第 8 章中应用程序中的有些不同。不再使用 `"/Session"`, 而是用 `"/myApp/Session"` 作为匹配模式。这是因为将 `Context` 实例的路径名设置为 `"/myApp"`。路径名会用来发送 `Cookie`, 只有当路径也是 `"/myApp"` 的时候, 浏览器才会将 `Cookie` 发送回服务器:

```
context.addServletMapping("/myApp/Session", "Session");
```

用来请求 `SessionServlet` 实例的 URL 如下所示:

```
http://localhost:8080/myApp/Session.
```

与第 8 章类似, 需要为 `Context` 容器添加一个监听器和一个载入器:

```
LifecycleListener listener = new SimpleContextConfig();
((Lifecycle) context).addLifecycleListener(listener);

// here is our loader
Loader loader = new WebappLoader();
// associate the loader with the Context
context.setLoader(loader);
connector.setContainer(context);
```

接下来是本章的重点内容, 这里使用了 `StandardManager` 的一个实例, 并将其与 `Context` 容器相关联:

```
Manager manager = new StandardManager();
context.setManager(manager);
```

最后, 初始化并启动连接器和 `Servlet` 容器:

```
connector.initialize();
((Lifecycle) connector).start();
((Lifecycle) context).start();
```

## 9.4.2 SimpleWrapperValve 类

回忆一下本章开头的内容, `Servlet` 实例可以调用 `javax.servlet.http.HttpServletRequest` 接口的 `getSession()` 方法获取 `Session` 对象。当调用 `getSession()` 方法时, `request` 对象必须调用与 `Context` 容器相关联的 `Session` 管理器。 `Session` 管理器组件要么创建一个新的 `session` 对象, 要么返回一个已经存在的 `session` 对象。 `request` 对象为了能够访问 `Session` 管理器, 它必须能够访问 `Context` 容器。为了达到此目的, 在 `SimpleWrapperValve` 类的 `invoke()` 方法中, 需要调用 `org.apache.catalina.Request` 接口的 `setContext()` 方法, 并传入 `Context` 实例。记住, `SimpleWrapperValve` 类的 `invoke()` 方法会调用被请求的 `Servlet` 实例的 `service()` 方法。因此, 必须在调用 `Servlet` 实例的 `service()` 方法调用前设置 `Context` 实例。代码清单 9-9 给出了 `SimpleWrapperValve` 类的 `invoke()` 方法的实现。高亮显示的代码是该类中新添加的。

代码清单 9-9 SimpleWrapperValve 类的 `invoke()` 方法的实现

```
public void invoke(Request request, Response response,
    ValveContext valveContext) throws IOException, ServletException {

    SimpleWrapper wrapper = (SimpleWrapper) getContainer();
    ServletRequest sreq = request.getRequest();
```

```

ServletResponse sres = response.getResponse();
Servlet servlet = null;
HttpServletRequest hreq = null;
if (sreq instanceof HttpServletRequest)
    hreq = (HttpServletRequest) sreq;
HttpServletRequest hres = null;
if (sres instanceof HttpServletResponse)
    hres = (HttpServletResponse) sres;

// pass the Context to the Request object so that
// the Request object can call the Manager
Context context = (Context) wrapper.getParent();
request.setContext(context);

// Allocate a servlet instance to process this request
try {
    servlet = wrapper.allocate();
    if (hres!=null && hreq!=null) {
        servlet.service(hreq, hres);
    }
    else {
        servlet.service(sreq, sres);
    }
}
catch (ServletException e) {
}
}

```

可以访问 Wrapper 实例，因此，可以调用 Container 接口的 getParent() 方法来获取 Context 实例。注意，Wrapper 实例被添加到了 Context 容器中。当获取 Context 实例后，就可以调用 Request 接口的 setContext() 方法来设置 Context 容器了。

正如本章开始处解释的，org.apache.catalina.connector.HttpRequestBase 类的私有方法 doGetSession() 会调用 Context 接口的 getManager() 方法来获取 Session 管理器对象：

```

// Return the requested session if it exists and is valid
Manager manager = null;
if (context != null)
    manager = context.getManager();

```

当获得了 Session 管理器对象后，就可以获取到 Session 对象了，或直接创建一个新的 Session 对象。

### 9.4.3 运行应用程序

要在 Windows 平台下运行该应用程序，需要在工作目录下执行如下命令：

```

java -classpath ./lib/servlet.jar;./lib/commons-collections.jar;./
ex09.pyrmont.startup.Bootstrap

```

要在 Linux 平台下运行，需要使用冒号来代替库文件之间的分号：

```

java -classpath ./lib/servlet.jar:./lib/commons-collection.jar:./
ex09.pyrmont.startup.Bootstap

```

要调用 servlet 实例 SessionServlet，可以在浏览器中输入如下 URL：



`http://localhost:8080/myApp/Session`

SessionServlet 实例使用一个 Session 对象存储一个值。该 servlet 实例会显示 Session 对象中存储的前一个值和当前的值。此外，它还会显示一个表单来供用户输入一个新值，如图 9-4 所示。

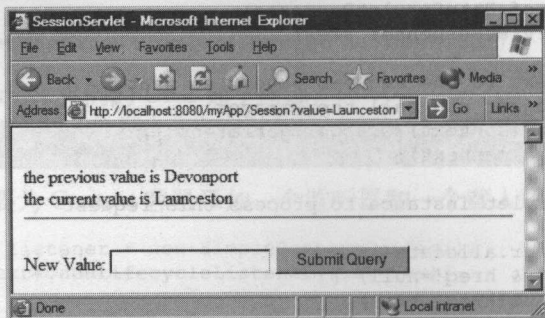


图 9-4 SessionServlet 类的输出

## 9.5 小结

本章讨论了 Session 管理器，该组件用来管理 Session 管理中的 Session 对象。解释了不同类型的 Session 管理器的区别，以及 Session 管理器是如何把 Session 对象持久化到存储器中。在本章的结尾处，通过一个应用程序来说明如何使用 StandardManager 实例来管理 Session 对象，并使用 Session 对象存储数据。

## 第 10 章

# 安全性

Web 应用程序的一些内容是受限的，只有授权的用户在提供了正确的用户名和密码后才能查看它们。servlet 技术支持通过配置部署描述器（web.xml 文件）来对这些内容进行访问控制。本章将学习 Web 容器是如何支持安全限制功能的。

servlet 容器是通过一个名为验证器的阀来支持安全限制的。当 servlet 容器启动时，验证器阀会被添加到 Context 容器的管道中。如果你忘记了 servlet 容器中管道的工作机制，请参见第 6 章。

在调用 Wrapper 阀之前，会先调用验证器阀，对当前用户进行身份验证。如果用户输入了正确的用户名和密码，则验证器阀会调用后续的阀，继续显示请求的 servlet。如果用户未能通过身份验证，则验证器阀会返回，而不会调用后面的阀。身份验证失败的话，用户就无法查看请求的 servlet 资源了。

验证器阀会调用 Context 容器的领域对象的 `authenticate()` 方法，传入用户输入的用户名和密码，来对用户进行身份验证。领域对象可以访问有效用户的用户名和密码的集合。

本章会首先介绍在 servlet 编程中表示与安全性功能有关的几个对象（如领域、主体和角色等）的类，然后用一个应用程序来说明如何使用基础身份验证来保护 servlet 资源。

**注意** 这里假设你已经熟悉了 servlet 编程中安全限制的概念，包括主体、角色、领域和登录配置等。如果你对 these 概念还不理解的话，可以阅读《Java for the Web with Senlets, JSP, and EJB》一书，或其他 servlet 编程的书籍。

### 10.1 领域

领域对象用来对用户进行身份验证的组件。它会对用户输入的用户名和密码对进行有效性判断。领域对象通常都会与一个 Context 容器相关联，而一个 Context 容器也只能有一个领域对象。可以调用 Context 容器的 `setRealm()` 方法来将领域对象与该 Context 容器相关联。

那么领域对象是如何验证用户身份的呢？实际上，它保存了所有有效用户的用户名和密码对，或者它会访问存储这些数据的存储器。这些数据的具体存储依赖于领域对象的具体实现。在 Tomcat 中，有效用户信息默认存储在 `tomcat-user.xml` 文件中。但是可以使用其他的领域对象的实现来针对其他资源验证用户身份，例如查询一个关系数据库。

在 Catalina 中，领域对象是 `org.apache.catalina.Realm` 接口的实例。该接口中有 4 个用来对用户进行身份验证的重载方法最为重要，其方法签名如下所示：

```

public Principal authenticate(String username, String credentials);
public Principal authenticate(String username, byte[] credentials);
public Principal authenticate(String username, String digest,
    String nonce, String nc, String cnonce, String qop, String realm,
    String md5a2);
public Principal authenticate(X509Certificate certs[]);

```

通常都会使用第 1 个重载方法。在 Realm 接口中，还有一个 hasRole() 方法，方法签名如下：

```
public boolean hasRole(Principal principal, String role);
```

此外，Realm 接口的 getContainer() 方法和 setContainer() 方法用来将 Realm 实例与一个 Context 实例相关联。

在 Catalina 中，Realm 接口的基本实现形式是 org.apache.catalina.realm.RealmBase 类，该类是一个抽象类。org.apache.catalina.realm 包还提供了 RealmBase 类一些继承类的实现，包括 JDBCRealm、JNDIRealm、MemoryRealm 和 UserDatabaseRealm 类等。默认情况下，会使用 MemoryRealm 类的实例作为验证用的领域对象。当第 1 次调用 MemoryRealm 实例时，它会读取 tomcat-users.xml 文档的内容。在本章的应用程序中，你会学习到如何建立一个简单的领域对象，并将用户身份信息保存在对象本身中。

**注意** 在 Catalina 中，验证器类会调用附加到其中的领域对象的 authenticate() 方法来验证用户身份。

## 10.2 GenericPrincipal 类

主体对象是 java.security.Principal 接口的实例。该接口在 Catalina 中的实现是 org.apache.catalina.realm.GenericPrincipal 类。GenericPrincipal 实例必须始终与一个领域对象相关联，正如 GenericPrincipal 的两个构造函数所示：

```

public GenericPrincipal(Realm realm, String name, String password) {
    this(realm, name, password, null);
}
public GenericPrincipal(Realm realm, String name, String password,
    List roles) {
    super();
    this.realm = realm;
    this.name = name;
    this.password = password;
    if (roles != null) {
        this.roles = new String[roles.size()];
        this.roles = (String[]) roles.toArray(this.roles);
        if (this.roles.length > 0)
            Arrays.sort(this.roles);
    }
}

```

GenericPrincipal 实例必须有 1 个用户名和密码对，此外，该用户名和密码对所对应的角色列表是可选的。然后，可以调用其 hasRole() 方法，并传入 1 个字符串形式的角色名来检查该主体对象是否拥有该指定角色。下面是 Tomcat 4 中 hasRole() 方法的实现。



```
public boolean hasRole(String role) {  
    if (role == null)  
        return (false);  
    return (Arrays.binarySearch(roles, role) >= 0);  
}
```

Tomcat 5 支持 Servlet 2.4 规范, 因此, 它必须能够识别特别字符 “\*”, 该字符表示任意角色:

```
public boolean hasRole(String role) {  
    if ("*".equals(role)) // Special 2.4 role meaning everyone  
        return true;  
    if (role == null)  
        return (false);  
    return (Arrays.binarySearch(roles, role) >= 0);  
}
```

### 10.3 LoginConfig 类

登录配置是 final 型的 org.apache.catalina.deploy.LoginConfig 类的实例, 其中包含 1 个领域对象的名字。LoginConfig 实例封装了领域对象名和所要使用的身份验证方法。可以调用 LoginConfig 实例的 getRealmName() 方法来获取领域对象的名字, 并调用其 getAuthName() 方法来获取所使用的身份验证方法的名字。获取的身份验证方法的名字必须是以下名字之一: BASIC、DIGEST、FORM 或 CLIENT-CERT。如果使用的是基于表单的身份验证方法, LoginConfig 实例还需要在 loginPage 属性和 errorPage 属性中分别存储字符串形式的登录页面和错误页面的 URL。

在实际部署中, Tomcat 在启动时需要读取 web.xml 文件的内容。如果 web.xml 文件包含 login-config 元素的配置, 则 Tomcat 会创建一个 LoginConfig 对象, 并设置其相应的属性。验证器阀会调用 LoginConfig 实例的 getRealmName() 方法获取领域对象名, 并将该领域对象名发送到浏览器, 显示在登录对话框中。如果 getRealmName() 方法的返回值为 null, 则会将服务器名和相应端口发送给浏览器。图 10-1 展示了在 Windows XP 系统中使用 Internet Explorer 6 浏览器进行基本身份验证的登录对话框。

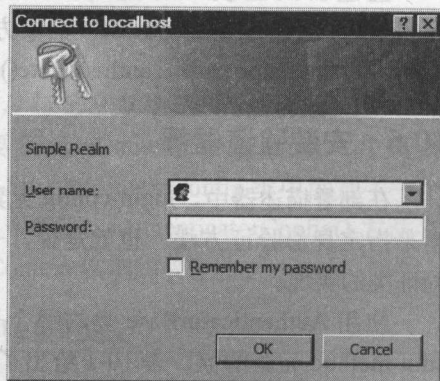


图 10-1 基本身份验证对话框

### 10.4 Authenticator 接口

验证器是 org.apache.catalina.Authenticator 接口的实例。Authenticator 接口本身并没有声明方法, 只是起到了一个标记的作用, 这样其他的组件就可以使用 instanceof 关键字检查某个组件是不是一个验证器。

Catalina 提供了 Authenticator 接口的一个基本实现, org.apache.catalina.authenticator.AuthenticatorBase 类。除了实现 Authenticator 接口外, AuthenticatorBase 类还扩展了 org.apache.catalina.valves.ValveBase 类。这就是说, AuthenticatorBase 类也是一个阀。在 org.apache.catalina.authenticator 包下有很多实现类, 包括 BasicAuthenticator 类 (可以用来支持基本的身份验证)、FormAuthenticator 类 (提供了基于表单的身份验证)、DigestAuthentication 类 (提供

了基于信息摘要的身份验证)和 SSLAuthenticator 类(用于对 SSL 进行身份验证)。此外,当 Tomcat 用户没有指定验证方法名时,NonLoginAuthenticator 类用于对来访者的身份进行验证。NonLoginAuthenticator 类实现的验证器只会检查安全限制,而不会涉及用户身份的验证。

图 10-2 给出了 org.apache.catalina.authenticator 包下各个功能类的 UML 类图。

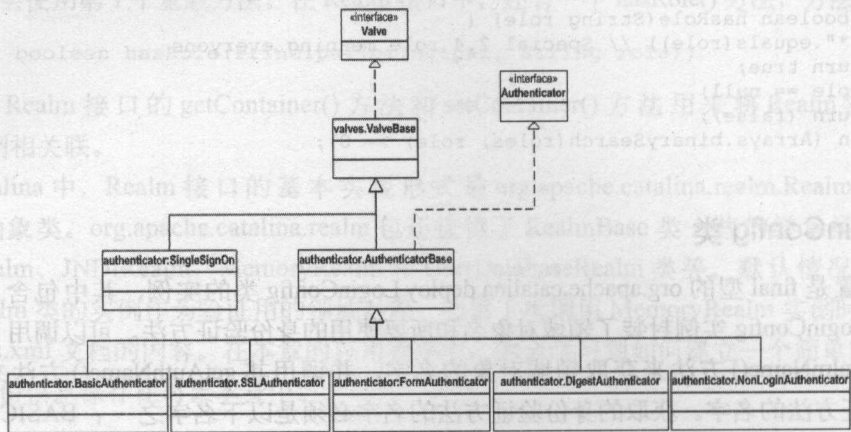


图 10-2 验证器相关类的 UML 类图

验证器的重要工作是对用户进行身份验证。因此,当你看到 AuthenticatorBase 类的 invoke() 方法调用 authenticate() 抽象方法时,后者的实现依赖于子类,请不要惊讶。例如,在 BasicAuthenticator 类中,authenticate() 方法会使用基本身份验证来验证用户的身份信息。

10.5 安装验证器阀

在部署描述器中,login-config 元素仅能出现一次。login-config 元素包含一个 auth-method 元素来指定身份验证方法。也就是说,一个 Context 实例只能有一个 LoginConfig 实例和利用一个验证类的实现。

使用 AuthenticatorBase 类的哪个子类作为 Context 实例中的验证器阀依赖于部署描述器中 auth-method 元素的值。表 10-1 给出了 auth-method 元素的值和与其对应的验证器的类名。

表 10-1 验证器的实现类

auth-method 元素的值	验证器类
BASIC	BasicAuthenticator
FORM	FormAuthenticator
DIGEST	DigestAuthenticator
CLIENT-CERT	SSLAuthenticator

若没有设置 auth-method 元素,则 LoginConfig 对象的 auth-method 属性的值默认为 NONE,这时会使用 NonLoginAuthenticator 进行安全验证。

由于使用的验证器类是在运行时才确定的,因此该类是动态载入的。StandardContext 类使

用 `org.apache.catalina.startup.ContextConfig` 类来对 `StandardContext` 实例的属性进行设置。这些设置包括实例化一个验证器类，并将该实例与 `Context` 实例相关联。本章的应用程序使用了 `ex10.pyrmont.core.SimpleContextConfig` 类来进行 `Context` 实例的属性设置。正如你将在后面小节中看到的，该类的实例负责动态载入 `BasicAuthenticator` 类，实例化其对象，并将其作为一个阀安装到 `StandardContext` 实例中。

**注意** `org.apache.catalina.startup.ContextConfig` 类的具体内容将在第 15 章讨论。

## 10.6 应用程序

本章的应用程序会使用 `Catalina` 中与安全相关的一些类。使用到的 `SimplePipeline` 类、`SimpleWrapper` 类、`SimpleContextConfig` 类和 `SimpleWrapperValve` 类与第 9 章中的应用程序类似。此外，`SimpleContextConfig` 类与第 9 章中的 `SimpleContextConfig` 类类似，除了 `SimpleContextConfig` 类的改动是该类有一个 `authenticatorConfig()` 方法，会将 `BasicAuthenticator` 实例添加到 `StandardContext` 实例中。本章的两个应用程序都使用了两个相同的测试 `Servlet`，分别是 `Primitiveservlet` 类和 `Modernservlet` 类。

第 1 个应用程序包含两个类：`ex10.pyrmont.startup.Bootstrap1` 类和 `ex10.pyrmont.realm.SimpleRealm` 类。第 2 个应用程序包含 `ex10.pyrmont.startup.Bootstrap2` 类和 `ex10.pyrmont.realm.SimpleUserDatabaseRealm` 类。下面几节将会对这些类详细讨论。

### 10.6.1 `ex10.pyrmont.core.SimpleContextConfig` 类

代码清单 10-1 给出了 `SimpleContextConfig` 类的定义，与第 9 章中的 `SimpleContextConfig` 类相似。`org.apache.catalina.core.StandardContext` 实例需要 `SimpleContextConfig` 实例，并将其 `configured` 属性设置为 `true`。但是，在本章的应用程序中，`SimpleContextConfig` 类中添加了一个 `authenticatorConfig()` 方法，该方法从 `lifeCycleEvent()` 方法中调用。`authenticatorConfig()` 方法实例化 `BasicAuthenticator` 类，并将其作为阀添加到 `StandardContext` 实例的管道中。

代码清单 10-1 `SimpleContextConfig` 类的定义

```
package ex10.pyrmont.core;

import org.apache.catalina.Authenticator;
import org.apache.catalina.Context;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleEvent;
import org.apache.catalina.LifecycleListener;
import org.apache.catalina.Pipeline;
import org.apache.catalina.Valve;
import org.apache.catalina.core.StandardContext;
import org.apache.catalina.deploy.SecurityConstraint;
import org.apache.catalina.deploy.LoginConfig;

public class SimpleContextConfig implements LifecycleListener {
    private Context context;
    public void lifecycleEvent(LifecycleEvent event) {
        if (Lifecycle.START_EVENT.equals(event.getType())) {
            context = (Context) event.getLifecycle();
        }
    }
}
```



```

        authenticatorConfig();
        context.setConfigured(true);
    }

    private synchronized void authenticatorConfig() {
        // Does this Context require an Authenticator?
        SecurityConstraint constraints[] = context.findConstraints();
        if ((constraints == null) || (constraints.length == 0))
            return;
        LoginConfig loginConfig = context.getLoginConfig();
        if (loginConfig == null) {
            loginConfig = new LoginConfig("NONE", null, null, null);
            context.setLoginConfig(loginConfig);
        }

        // Has an authenticator been configured already?
        Pipeline pipeline = ((StandardContext) context).getPipeline();
        if (pipeline != null) {
            Valve basic = pipeline.getBasic();
            if ((basic != null) && (basic instanceof Authenticator))
                return;
            Valve valves[] = pipeline.getValves();
            for (int i = 0; i < valves.length; i++) {
                if (valves[i] instanceof Authenticator)
                    return;
            }
        }
        else { // no Pipeline, cannot install authenticator valve
            return;
        }
        // Has a Realm been configured for us to authenticate against?
        if (context.getRealm() == null) {
            return;
        }
        // Identify the class name of the Valve we should configure
        String authenticatorName =
            "org.apache.catalina.authenticator.BasicAuthenticator";
        // Instantiate and install an Authenticator of the requested class
        Valve authenticator = null;
        try {
            Class authenticatorClass = Class.forName(authenticatorName);
            authenticator = (Valve) authenticatorClass.newInstance();
            ((StandardContext) context).addValve(authenticator);
            System.out.println("Added authenticator valve to Context");
        }
        catch (Throwable t) {
        }
    }
}

```

`authenticatorConfig()` 方法会先检查在相关联的 Context 容器是否有安全限制。如果没有，该方法会直接返回，而不会安装验证器：

```

// Does this Context require an Authenticator?
SecurityConstraint constraints[] = context.findConstraints();
if ((constraints == null) || (constraints.length == 0))
    return;

```

如果当前 Context 容器中有一个或多个安全限制，`authenticatorConfig()` 方法会检查该

Context 实例是否有 LoginConfig 对象。若没有，则它会创建一个新的 LoginConfig 实例：

```
LoginConfig loginConfig = context.getLoginConfig();
if (loginConfig == null) {
    loginConfig = new LoginConfig("NONE", null, null, null);
    context.setLoginConfig(loginConfig);
}
```

然后，authenticatorConfig() 方法会检查当前 StandardContext 对象的管道中的基础阀或附加阀是否是验证器。因为一个 Context 实例只能有一个验证器，所以当发现某个阀是验证器后，authenticatorConfig() 方法就会直接返回：

```
// Has an authenticator been configured already?
Pipeline pipeline = ((StandardContext) context).getPipeline();
if (pipeline != null) {
    Valve basic = pipeline.getBasic();
    if ((basic != null) && (basic instanceof Authenticator))
        return;
    Valve valves[] = pipeline.getValves();
    for (int i = 0; i < valves.length; i++) {
        if (valves[i] instanceof Authenticator)
            return;
    }
}
else { // no Pipeline, cannot install authenticator valve
    return;
}
```

然后，它会检查当前 Context 实例中是否有与之关联的领域对象。如果没有找到领域对象，就不需要安装验证器了，因为用户是无法通过身份验证的：

```
// Has a Realm been configured for us to authenticate against?
if (context.getRealm() == null) {
    return;
}
```

若找到了领域对象，则 authenticatorConfig() 方法会动态载入 BasicAuthenticator 类，创建该类的一个实例，并将其作为阀添加到 StandardContext 实例中：

```
// Identify the class name of the Valve we should configure
String authenticatorName =
    "org.apache.catalina.authenticator.BasicAuthenticator";
// Instantiate and install an Authenticator of the requested class
Valve authenticator = null;
try {
    Class authenticatorClass = Class.forName(authenticatorName);
    authenticator = (Valve) authenticatorClass.newInstance();
    ((StandardContext) context).addValve(authenticator);
    System.out.println("Added authenticator valve to Context");
}
catch (Throwable t) { }
```

## 10.6.2 ex10.pyrmont.realm.SimpleRealm 类

代码清单 10-2 给出了 SimpleRealm 类的定义，该类说明了领域对象是如何工作的。该类在本章的第 1 个应用程序中使用，包含两个硬编码的用户名和密码对。

## 代码清单 10-2 SimpleRealm 类的定义

```

package ex10.pyrmont.realm;

import java.beans.PropertyChangeListener;
import java.security.Principal;
import java.security.cert.X509Certificate;
import java.util.ArrayList;
import java.util.Iterator;
import org.apache.catalina.Container;
import org.apache.catalina.Realm;
import org.apache.catalina.realm.GenericPrincipal;

public class SimpleRealm implements Realm {
    public SimpleRealm() {
        createUserDataBase();
    }
    private Container container;
    private ArrayList users = new ArrayList();
    public Container getContainer() {
        return container;
    }
    public void setContainer(Container container) {
        this.container = container;
    }
    public String getInfo() {
        return "A simple Realm implementation";
    }
    public void addPropertyChangeListener(PropertyChangeListener
        listener) { }
    public Principal authenticate(String username, String credentials) {
        System.out.println("SimpleRealm.authenticate()");
        if (username==null || credentials==null)
            return null;
        User user = getUser(username, credentials);
        if (user==null)
            return null;
        return new GenericPrincipal(this, user.username,
            user.password, user.getRoles());
    }
    public Principal authenticate(String username, byte[] credentials) {
        return null;
    }
    public Principal authenticate(String username, String digest,
        String nonce, String nc, String cnonce, String qop, String realm,
        String md5a2) {
        return null;
    }
    public Principal authenticate(X509Certificate certs[]) {
        return null;
    }
    public boolean hasRole(Principal principal, String role) {
        if ((principal == null) || (role == null) ||
            !(principal instanceof GenericPrincipal))
            return (false);
        GenericPrincipal gp = (GenericPrincipal) principal;
        if (!(gp.getRealm() == this))
            return (false);
        boolean result = gp.hasRole(role);
        return result;
    }
}

```



```

public void removePropertyChangeListener(PropertyChangeListener
    listener) { }
private User getUser(String username, String password) {
    Iterator iterator = users.iterator();
    while (iterator.hasNext()) {
        User user = (User) iterator.next();
        if (user.username.equals(username) &&
            user.password.equals(password))
            return user;
    }
    return null;
}
private void createUserDatabase() {
    User user1 = new User("ken", "blackcomb");
    user1.addRole("manager");
    user1.addRole("programmer");
    User user2 = new User("cindy", "bamboo");
    user2.addRole("programmer");

    users.add(user1);
    users.add(user2);
}

class User {
    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }
    public String username;
    public ArrayList roles = new ArrayList();
    public String password;
    public void addRole(String role) {
        roles.add(role);
    }
    public ArrayList getRoles() {
        return roles;
    }
}
}

```

SimpleRealm 类实现 Realm 接口。在构造函数中，它会调用 createUserDatabase() 方法创建两个用户。在内部，用户是内部类 User 的实例。第 1 个用户的用户名为 ken，密码为 blackcomb。该用户有两个角色：管理员角色和程序员角色。第 2 个用户的用户名和密码分别是 cindy 和 bamboo。该用户只有程序员角色。然后，将这两个用户添加到 ArrayList 类型的变量 users 中。创建两个用户的代码如下所示：

```

User user1 = new User("ken", "blackcomb");
user1.addRole("manager");
user1.addRole("programmer");
User user2 = new User("cindy", "bamboo");
user2.addRole("programmer");
users.add(user1);
users.add(user2);

```

SimpleRealm 类提供了 authenticate() 方法的 4 个重载版本中的一个实现：

```

public Principal authenticate(String username, String credentials) {
    System.out.println("SimpleRealm.authenticate()");
    if (username==null || credentials==null)
        return null;
    User user = getUser(username, credentials);
    if (user==null)
        return null;
    return new GenericPrincipal(this, user.username,
        user.password, user.getRoles());
}

```

authenticate() 方法由验证器调用。如果用户提供的用户名和密码是无效的，该方法会返回 null。否则，它会返回一个代表该用户的 Principal 对象。

### 10.6.3 ex10.pyrmont.realm.SimpleUserDatabaseRealm

SimpleUserDatabaseRealm 类表示一个复杂一点的领域对象，它并不将用户列表存储到对象自身中。相反，它会读取 conf 目录下的 tomcat-users.xml 文件，将内容载入内存。然后，依据该列表进行身份验证工作。在本书附带的 ZIP 文件内的 conf 目录中，有 tomcat-users.xml 文件的一个副本，内容如下：

```

<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
    <role rolename="tomcat"/>
    <role rolename="role1"/>
    <role rolename="manager"/>
    <role rolename="admin"/>
    <user username="tomcat" password="tomcat" roles="tomcat"/>
    <user username="role1" password="tomcat" roles="role1"/>
    <user username="both" password="tomcat" roles="tomcat,role1"/>
    <user username="admin" password="password" roles="admin,manager"/>
</tomcat-users>

```

代码清单 10-3 给出了 SimpleUserDatabaseRealm 类的定义。

代码清单 10-3 SimpleUserDatabaseRealm 类的定义

```

package ex10.pyrmont.realm;
// modification of org.apache.catalina.realm.UserDatabaseRealm
import java.security.Principal;
import java.util.ArrayList;
import java.util.Iterator;
import org.apache.catalina.Group;
import org.apache.catalina.Role;
import org.apache.catalina.User;
import org.apache.catalina.realm.UserDatabase;
import org.apache.catalina.realm.GenericPrincipal;
import org.apache.catalina.realm.RealmBase;
import org.apache.catalina.users.MemoryUserDatabase;

public class SimpleUserDatabaseRealm extends RealmBase {
    protected UserDatabase database = null;
    protected static final String name = "SimpleUserDatabaseRealm";
    protected String resourceName = "UserDatabase";
    public Principal authenticate(String username, String credentials) {
        // Does a user with this username exist?
        User user = database.findUser(username);
        if (user == null) {

```

```

return (null);
}
// Do the credentials specified by the user match?
boolean validated = false;
if (hasMessageDigest()) {
    // Hex hashes should be compared case-insensitive
    validated =
        digest(credentials).equalsIgnoreCase(user.getPassword());
}
else {
    validated = (digest(credentials).equals(user.getPassword()));
}
if (!validated) {
    return null;
}

ArrayList combined = new ArrayList();
Iterator roles = user.getRoles();
while (roles.hasNext()) {
    Role role = (Role) roles.next();
    String rolename = role.getRolename();
    if (!combined.contains(rolename)) {
        combined.add(rolename);
    }
}
Iterator groups = user.getGroups();
while (groups.hasNext()) {
    Group group = (Group) groups.next();
    roles = group.getRoles();
    while (roles.hasNext()) {
        Role role = (Role) roles.next();
        String rolename = role.getRolename();
        if (!combined.contains(rolename)) {
            combined.add(rolename);
        }
    }
}
return (new GenericPrincipal(this, user.getUsername(),
    user.getPassword(), combined));
}

protected Principal getPrincipal(String username) {
    return (null);
}

protected String getPassword(String username) {
    return null;
}

protected String getName() {
    return this.name;
}

public void createDatabase(String path) {
    database = new MemoryUserDatabase(name);
    ((MemoryUserDatabase) database).setPathname(path);
    try {
        database.open();
    }
    catch (Exception e) {
    }
}
}

```



在实例化 SimpleUserDatabaseRealm 类后，必须调用它的 createDatabase() 方法，并向包含用户列表的 XML 文档传递路径。createDatabase() 方法会实例化 org.apache.catalina.users.MemoryUserDatabase 类，后者读取并解析 XML 文档的内容。

#### 10.6.4 ex10.pyrmont.startup.Bootstrap1 类

Bootstrap1 类用于启动本章中的第 1 个应用程序。代码清单 10-4 给出了 Bootstrap1 类的定义。

代码清单 10-4 Bootstrap1 类的定义

```
package ex10.pyrmont.startup;

import ex10.pyrmont.core.SimpleWrapper;
import ex10.pyrmont.core.SimpleContextConfig;
import ex10.pyrmont.realm.SimpleRealm;
import org.apache.catalina.Connector;
import org.apache.catalina.Context;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleListener;
import org.apache.catalina.Loader;
import org.apache.catalina.Realm;
import org.apache.catalina.Wrapper;
import org.apache.catalina.connector.http.HttpConnector;
import org.apache.catalina.core.StandardContext;
import org.apache.catalina.deploy.LoginConfig;
import org.apache.catalina.deploy.SecurityCollection;
import org.apache.catalina.deploy.SecurityConstraint;
import org.apache.catalina.loader.WebappLoader;

public final class Bootstrap1 {
    public static void main(String[] args) {
        System.setProperty("catalina.base",
            System.getProperty("user.dir"));
        Connector connector = new HttpConnector();
        Wrapper wrapper1 = new SimpleWrapper();
        wrapper1.setName("Primitive");
        wrapper1.setServletClass("PrimitiveServlet");
        Wrapper wrapper2 = new SimpleWrapper();
        wrapper2.setName("Modern");
        wrapper2.setServletClass("ModernServlet");

        Context context = new StandardContext();
        // StandardContext's start method adds a default mapper
        context.setPath("/myApp");
        context.setDocBase("myApp");
        LifecycleListener listener = new SimpleContextConfig();
        ((Lifecycle) context).addLifecycleListener(listener);

        context.addChild(wrapper1);
        context.addChild(wrapper2);
        // for simplicity, we don't add a valve, but you can add
        // valves to context or wrapper just as you did in Chapter 6

        Loader loader = new WebappLoader();
        context.setLoader(loader);
        // context.addServletMapping(pattern, name);
        context.addServletMapping("/Primitive", "Primitive");
        context.addServletMapping("/Modern", "Modern");
        // add ContextConfig. This listener is important because it
```

```
// configures StandardContext (sets configured to true), otherwise
// StandardContext won't start
```

```
// add constraint
```

```
SecurityCollection securityCollection = new SecurityCollection();
securityCollection.addPattern("/");
securityCollection.addMethod("GET");
```

```
SecurityConstraint constraint = new SecurityConstraint();
```

```
constraint.addCollection(securityCollection);
```

```
constraint.addAuthRole("manager");
```

```
LoginConfig loginConfig = new LoginConfig();
```

```
loginConfig.setRealmName("Simple Realm");
```

```
// add realm
```

```
Realm realm = new SimpleRealm();
```

```
context.setRealm(realm);
```

```
context.addConstraint(constraint);
```

```
context.setLoginConfig(loginConfig);
```

```
connector.setContainer(context);
```

```
try {
```

```
    connector.initialize();
```

```
    ((Lifecycle) connector).start();
```

```
    ((Lifecycle) context).start();
```

```
    // make the application wait until we press a key.
```

```
    System.in.read();
```

```
    ((Lifecycle) context).stop();
```

```
} catch (Exception e) {
```

```
    e.printStackTrace();
```

```
}
```

```
}
```

Bootstrap1 类的主方法会创建两个 SimpleWrapper 对象，并调用其 setName() 方法和 setServletClass() 方法。对于第 1 个 SimpleWrapper 对象，会把“Primitive”传入到其 setName() 方法中，把“PrimitiveServlet”传入到其 setServletClass() 方法中。相应地，第 2 个 SimpleWrapper 对象则会分别获取“Modern”和“ModernServlet”。

然后，main() 方法会创建一个 StandardContext 对象，设置其 path 属性和 documentBase 属性，并添加一个 SimpleContextConfig 类的监听器。记住，后者会把一个 BasicAuthenticator 对象安装到 StandardContext 对象中。接下来，它会向 StandardContext 添加一个载入器并添加两个 servlet 映射。在第 9 章的应用程序中有与此相同的代码。下面是本章中新增加的一些代码：

```
// add constraint
```

```
SecurityCollection securityCollection = new SecurityCollection();
```

```
securityCollection.addPattern("/");
```

```
securityCollection.addMethod("GET");
```

main() 方法会创建一个 SecurityCollection 对象，并调用其 addPattern() 方法和 addMethod() 方法。addPattern() 方法指定某个 URL 要遵循哪个安全限制。addMethod() 方法会指定该安全限制要使用哪种验证方法。在 addMethod() 方法中设置为“GET”，所以，使用 GET 方法提交的

HTTP 请求会遵循安全限制。

接下来, `main()` 方法会实例化一个 `SecurityConstraint` 对象, 并将其添加到安全限制集合中。此外, 它还会设置哪种角色可以访问受限资源。在程序代码中, 传入了字符串 “manager”, 因此, 具有管理员角色的用户就可以访问这些受限资源。注意, 在 `SimpleRealm` 类的定义中只有一个用户有管理员角色, 也就是用户 `ken`, 它的密码是 `blackcomb`:

```
SecurityConstraint constraint = new SecurityConstraint();
constraint.addCollection(securityCollection);
constraint.addAuthRole("manager");
```

接下来, `main()` 方法会创建一个 `LoginConfig` 对象和一个 `SimpleRealm` 对象:

```
LoginConfig loginConfig = new LoginConfig();
loginConfig.setRealmName("Simple Realm");
// add realm
Realm realm = new SimpleRealm();
```

然后, 它会将领域对象、安全限制对象、登录配置对象与 `StandardContext` 实例相关联:

```
context.setRealm(realm);
context.addConstraint(constraint);
context.setLoginConfig(loginConfig);
```

接着, 它启动 `Context` 实例。这部分内容已经在前几章中介绍过了, 这里不再赘述。

实际上, 对 `PrimitiveServlet` 类和 `ModernServlet` 类的访问是受限的。如果用户请求访问其中的某个 `servlet` 资源, 则使用基本身份验证对其身份进行验证。只有用户提供了正确的用户名和密码 (在这里, 必须是 `ken` 和 `blackcomb`) 后, 才允许其访问该资源。

### 10.6.5 ex10.pyrmont.startup.Bootstrap2 类

`Bootstrap2` 类用于启动本章中的第 2 个应用程序。该类与 `Bootstrap1` 类相似, 区别在于 `Bootstrap2` 类使用了 `SimpleUserDatabase` 类的一个实例作为与 `StandardContext` 实例相关联的领域对象。要访问 `PrimitiveServlet` 类和 `ModernServlet` 类, 用户必须输入正确的用户名和密码。

代码清单 10-5 给出了 `Bootstrap2` 类的定义。

代码清单 10-5 `Bootstrap2` 类的定义

```
package ex10.pyrmont.startup;

import ex10.pyrmont.core.SimpleWrapper;
import ex10.pyrmont.core.SimpleContextConfig;
import ex10.pyrmont.realm.SimpleUserDatabaseRealm;
import org.apache.catalina.Connector;
import org.apache.catalina.Context;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleListener;
import org.apache.catalina.Loader;
import org.apache.catalina.Realm;
import org.apache.catalina.Wrapper;
import org.apache.catalina.connector.http.HttpConnector;
import org.apache.catalina.core.StandardContext;
import org.apache.catalina.deploy.LoginConfig;
import org.apache.catalina.deploy.SecurityCollection;
import org.apache.catalina.deploy.SecurityConstraint;
import org.apache.catalina.loader.WebappLoader;
```



```
public final class Bootstrap2 {
    public static void main(String[] args) {
        System.setProperty("catalina.base",
            System.getProperty("user.dir"));
        Connector connector = new HttpConnector();
        Wrapper wrapper1 = new SimpleWrapper();
        wrapper1.setName("Primitive");
        wrapper1.setServletClass("PrimitiveServlet");
        Wrapper wrapper2 = new SimpleWrapper();
        wrapper2.setName("Modern");
        wrapper2.setServletClass("ModernServlet");

        Context context = new StandardContext();
        // StandardContext's start method adds a default mapper
        context.setPath("/myApp");
        context.setDocBase("myApp");
        LifecycleListener listener = new SimpleContextConfig();
        ((Lifecycle) context).addLifecycleListener(listener);

        context.addChild(wrapper1);
        context.addChild(wrapper2);
        // for simplicity, we don't add a valve, but you can add
        // valves to context or wrapper just as you did in Chapter 6

        Loader loader = new WebappLoader();
        context.setLoader(loader);
        // context.addServletMapping(pattern, name);
        context.addServletMapping("/Primitive", "Primitive");
        context.addServletMapping("/Modern", "Modern");
        // add ContextConfig. This listener is important because it
        // configures StandardContext (sets configured to true), otherwise
        // StandardContext won't start

        // add constraint
        SecurityCollection securityCollection = new SecurityCollection();
        securityCollection.addPattern("/");
        securityCollection.addMethod("GET");

        SecurityConstraint constraint = new SecurityConstraint();
        constraint.addCollection(securityCollection);
        constraint.addAuthRole("manager");
        LoginConfig loginConfig = new LoginConfig();
        loginConfig.setRealmName("Simple User Database Realm");
        // add realm
        Realm realm = new SimpleUserDatabaseRealm();
        ((SimpleUserDatabaseRealm) realm).
            createDatabase("conf/tomcat-users.xml");
        context.setRealm(realm);
        context.addConstraint(constraint);
        context.setLoginConfig(loginConfig);

        connector.setContainer(context);

        try {
            connector.initialize();
            ((Lifecycle) connector).start();
            ((Lifecycle) context).start();

            // make the application wait until we press a key.
            System.in.read();
            ((Lifecycle) context).stop();
        }
    }
}
```

```
catch (Exception e) {  
    e.printStackTrace();  
}  
}
```

### 10.6.6 运行应用程序

要在 Windows 平台下运行第 1 个应用程序，需要在工作目录中执行以下命令：

```
java -classpath ./lib/servlet.jar;./lib/commons-collections.jar;./  
ex10.pyrmont.startup.Bootstrap1
```

而在 Linux 平台下，需要使用冒号来分隔两个库：

```
java -classpath ./lib/servlet.jar:./lib/commons-collections.jar:./  
ex10.pyrmont.startup.Bootstrap1
```

要在 Windows 平台下运行第 2 个应用程序，需要在工作目录中执行以下命令：

```
java -classpath ./lib/servlet.jar;./lib/commons-collections.jar;  
./lib/commons-digester.jar;./lib/commons-logging.jar;./  
ex10.pyrmont.startup.Bootstrap2
```

而在 Linux 平台下，需要使用冒号来分隔两个库：

```
java -classpath ./lib/servlet.jar:./lib/commons-collections.jar:  
./lib/commons-digester.jar:./lib/commons-logging.jar:./  
ex10.pyrmont.startup.Bootstrap2
```

要在两个应用程序中调用 Primitive servlet，需要在浏览器中输入以下 URL：

```
http://localhost:8080/Primitive
```

要在应用程序中访问 Modern servlet，需要在浏览器中输入以下 URL：

```
http://localhost:8080/ Modern
```

## 10.7 小结

在 servlet 编程中，安全性是一个非常重要的话题。servlet 规范通过提供一些与安全相关的对象（如主体对象、角色、安全限制和登录配置等）来满足对安全性的需要。在本章中，你已经学会了 servlet 容器是如何解决安全验证问题的。

## 第 11 章

# StandardWrapper

第 5 章已经说明了 Tomcat 中有 4 种类型的 servlet 容器，分别是 Engine、Host、Context 和 Wrapper。在前面的章节中你已经学会了如何构建一个简单的 Context 容器和 Wrapper 容器。一般情况下，Context 容器包含一个或多个 Wrapper 实例，每个 Wrapper 实例表示一个具体的 servlet 定义。本章将要对 Catalina 中 Wrapper 接口的标准实现进行说明。本章首先会对处理 HTTP 请求的方法序列进行说明，然后介绍 javax.servlet.SingleThreadModel 接口，最后是对 StandardWrapper 类和 StandardWrapperValve 类的说明。本章的应用程序会使用 StandardWrapper 实例来表示一个具体的 servlet 定义。

### 11.1 方法调用序列

对于每个引入的 HTTP 请求，连接器都会调用与其关联的 servlet 容器的 invoke() 方法。然后，servlet 容器会调用其所有子容器的 invoke() 方法。例如，若连接器与一个 StandardContext 实例相关联，则连接器会调用 StandardContext 实例的 invoke() 方法，而 StandardContext 实例会调用其所有子容器的 invoke() 方法（在本例中，就是调用 StandardWrapper 的 invoke() 方法）。图 11-1 展示了连接器接收到 HTTP 请求后的方法调用的协作图（回忆一下第 5 章的内容，servlet 容器包含一条管道和一个或多个阀）。

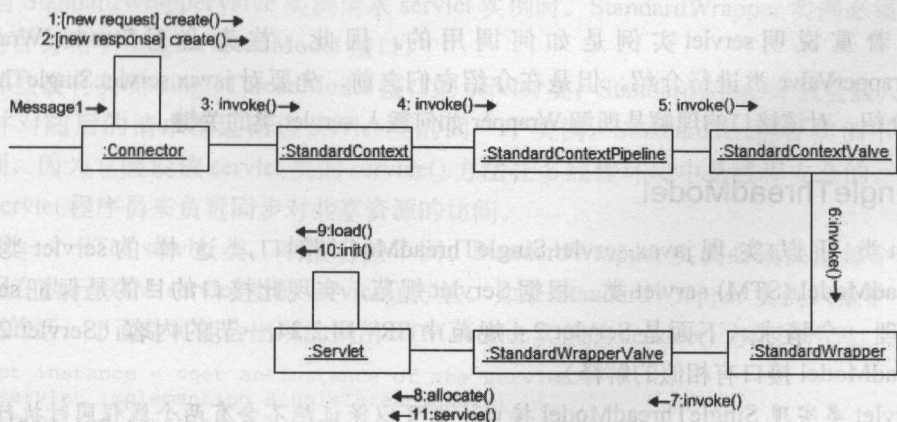


图 11-1 处理 HTTP 请求的方法调用协作图

具体过程如下：

- 1) 连接器创建 request 和 response 对象；



2) 连接器调用 StandardContext 实例的 invoke() 方法;

3) 接着, StandardContext 实例的 invoke() 方法调用其管道对象的 invoke() 方法。StandardContext 中管道对象的基础阀是 StandardContextValve 类的实例, 因此, StandardContext 的管道对象会调用 StandardContextValve 实例的 invoke() 方法;

4) StandardContextValve 实例的 invoke() 方法获取相应的 Wrapper 实例处理 HTTP 请求, 调用 Wrapper 实例的 invoke() 方法;

5) StandardWrapper 类是 Wrapper 接口的标准实现, StandardWrapper 实例的 invoke() 方法会调用其管道对象的 invoke() 方法;

6) StandardWrapper 的管道对象中的基础阀是 StandardWrapperValve 类的实例, 因此, 会调用 StandardWrapperValve 的 invoke() 方法, StandardWrapperValve 的 invoke() 方法调用 Wrapper 实例的 allocate() 方法获取 servlet 实例;

7) allocate() 方法调用 load() 方法载入相应的 servlet 类, 若已经载入, 则无需重复载入;

8) load() 方法调用 servlet 实例的 init() 方法;

9) StandardWrapperValve 调用 servlet 实例的 service() 方法。

**注意** StandardContext 类的构造函数会设置 StandardContextValve 类的一个实例作为其基础阀:

```
public StandardContext () {  
    super();  
    pipeline.setBasic(new StandardContextValve());  
    namingResources.setContainer(this);  
}
```

**注意** StandardWrapper 类的构造函数也会设置一个 StandardWrapperValve 实例作为其基础阀:

```
public StandardWrapper () {  
    super();  
    pipeline.setBasic(new StandardWrapperValve());  
}
```

本章着重说明 servlet 实例是如何调用的。因此, 首先会对 StandardWrapper 类和 StandardWrapperValve 类进行介绍。但是在介绍它们之前, 先要对 javax.servlet.SingleThreadModel 接口进行介绍。对该接口的理解是理解 Wrapper 如何载入 servlet 类的关键。

## 11.2 SingleThreadModel

servlet 类可以实现 javax.servlet.SingleThreadModel 接口, 这样的 servlet 类也称为 SingleThreadModel (STM) servlet 类。根据 Servlet 规范, 实现此接口的目的是保证 servlet 实例一次只处理一个请求。下面是 Servlet 2.4 规范中 SRV.14.2.24 一节的内容 (Servlet 2.3 规范对 SingleThreadModel 接口有相似的解释):

若 servlet 类实现 SingleThreadModel 接口, 则可以保证绝不会有二个线程同时执行该 servlet 实例的 service() 方法。这一点由 servlet 容器通过控制对单一 servlet 实例的同步访问实现, 或者维护一个 servlet 实例池, 然后将每个新请求分派给一个空闲的 servlet 实例。该接口并不能防止 servlet 访问共享资源造成的同步问题, 例如访问类的静态变量或访问 servlet 作用域之外的类。

很多程序员并没有仔细阅读这段话, 想当然地认为, 实现了该接口的 servlet 就是线程安全

的。这种想法是错误的，请再读一遍上面的引文内容。

事实上，实现了 `SingleThreadModel` 接口的 `servlet` 类只能保证在同一时刻，只有一个线程在执行该 `servlet` 实例的 `service()` 方法。但是，为了提高执行性能，`servlet` 容器会创建多个 STM `servlet` 实例。也就是说，STM `servlet` 实例的 `service()` 方法会在多个 STM `servlet` 实例中并发执行。如果 `servlet` 实例需要访问静态类变量或类外的某些资源的话，就有可能引起同步问题。

#### 多线程的虚假安全性：

在 Servlet 2.4 规范中，`SingleThreadModel` 接口已经弃用了，因为它会使 `servlet` 程序员误以为实现了该接口的 `servlet` 类就是多线程安全的。但是，Servlet 2.3 规范和 Servlet 2.4 规范都还对该接口提供了支持。

**注意** 这里有一些有关于 `SingleThreadModel` 的讨论，有兴趣的话可以参阅：

<http://w4.metronet.com/~wjm/tomcat/ToFeb11/msg02655.html>

## 11.3 StandardWrapper

`StandardWrapper` 对象的主要任务是载入它所代表的 `servlet` 类，并进行实例化。但是，`StandardWrapper` 类并不调用 `servlet` 的 `service` 方法。该任务由 `StandardWrapperValve` 对象（`StandardWrapper` 实例的管道对象中的基础阀）完成。`StandardWrapperValve` 对象通过调用 `allocate()` 方法从 `StandardWrapper` 实例中获取 `servlet` 实例。在获得了 `servlet` 实例后，`StandardWrapperValve` 实例就会调用 `servlet` 实例的 `service()` 方法。

当第一次请求某个 `servlet` 类时，`StandardWrapper` 载入 `servlet` 类。由于 `StandardWrapper` 实例会动态地载入该 `servlet` 类，因此，它必须知道该 `servlet` 类的完全限定名。可以调用 `StandardWrapper` 的 `setServletClass()` 方法并指定该 `servlet` 类的完全限定名，也可以调用其 `setName()` 方法为该 `servlet` 类指定一个名字。

至于当 `StandardWrapperValve` 实例请求 `servlet` 实例时，`StandardWrapper` 实例必须考虑到该 `servlet` 类是否实现了 `SingleThreadModel` 接口。

对于那些没有实现 `SingleThreadModel` 接口的 `servlet` 类，`StandardWrapper` 只会载入该 `servlet` 类一次，并对随后的请求都返回该 `servlet` 类的同一个实例。`StandardWrapper` 实例不需要多个 `servlet` 实例，因为它假设该 `servlet` 类的 `service()` 方法在多线程环境中是线程安全的。如果必要的话，由 `servlet` 程序员来负责同步对共享资源的访问。

而对于一个 STM `servlet` 类，事情有些不同。`StandardWrapper` 实例必须保证每个时刻只能有一个线程在执行 STM `servlet` 类的 `service()` 方法。如果 `StandardWrapper` 实例只维护一个 STM `servlet` 实例的话，下面是可能会出现的调用 STM `servlet` 实例的 `service()` 方法的代码：

```
Servlet instance = <get an instance of the servlet>;
if ((servlet implementing SingleThreadModel) {
    synchronized (instance) {
        instance.service(request, response);
    }
}
else {
    instance.service(request, response);
}
```

但是, 为了获得更好的性能, StandardWrapper 实例会维护一个 STM servlet 实例池。

Wrapper 实例负责准备一个 javax.servlet.ServletConfig 实例, 后者在 servlet 实例内部可以获取到。接下来的两节会讨论如何分配并载入一个 servlet 类。

### 11.3.1 分配 servlet 实例

正如本节开头所述, StandardWrapperValve 实例的 invoke() 方法调用 Wrapper 实例的 allocate() 方法获取请求的 servlet 的一个实例。因此, StandardWrapper 类要实现 allocate() 方法。

该方法的签名如下:

```
public javax.servlet.Servlet allocate() throws ServletException;
```

注意, allocate() 方法返回请求的 servlet 的一个实例。

为了支持 STM servlet, allocate() 方法需要变得复杂一些。事实上, 为了处理 STM servlet 和非 STM servlet, allocate() 方法分为两个部分。第一部分的结构如下:

```
if (!singleThreadModel) {
    // returns a non-STM servlet instance
}
```

布尔变量 singleThreadModel 用来表明该 StandardWrapper 实例表示的 servlet 类是否是 STM servlet。该变量的初始值为 false, loadServlet() 方法会检查它正在载入的 servlet 类是不是一个 STM servlet 类, 并根据结果修改变量 singleThreadModel 的值。loadServlet() 方法的实现将在 11.3.2 节中介绍。

当布尔变量 singleThreadModel 的值为 true 时, 执行 allocate() 方法的第二部分, 其结构如下:

```
synchronized (instancePool) {
    // returns an instance of the servlet from the pool
}
```

下面看一下第一部分和第二部分。

对于非 STM servlet 类, StandardWrapper 类定义了一个名为 instance, 类型为 javax.servlet.Servlet 的变量:

```
private Servlet instance = null;
```

allocate() 方法会检查变量 instance 是否是 null。若是, 则 allocate() 方法调用 loadServlet() 方法载入相关的 servlet 类, 然后, 将整型变量 countAllocated 的值加 1, 并返回 instance 的值。代码如下:

```
if (!singleThreadModel) {
    // Load and initialize our instance if necessary
    if (instance == null) {
        synchronized (this) {
            if (instance == null) {
                try {
                    instance = loadServlet();
                }
                catch (ServletException e) {
                    throw e;
                }
                catch (Throwable e) {
                    throw new ServletException

```



```
(sm.getString("standardWrapper.allocate"), e);
```

```
if (!singleThreadModel) {
    if (debug >= 2)
        log(" Returning non-STM instance");
    countAllocated++;
    return (instance);
}
```

若 StandardWrapper 表示的 servlet 类是一个 STM servlet 类，则 allocate() 方法会试图从对象池中返回一个 servlet 实例。变量 instancePool 是一个 java.util.Stack 类型的栈，其中保存了所有的 STM servlet 实例：

```
private Stack instancePool = null;
```

该变量在 loadServlet() 方法中初始化，将在 11.3.2 节中讨论。

只要 STM servlet 实例数不超过指定的最大值，allocate() 方法会返回一个 STM servlet 实例。整型变量 maxInstances 保存了在栈中存储的 STM servlet 实例的最大值，默认值是 20：

```
private int maxInstances = 20;
```

为了跟踪当前 STM servlet 实例的数量，StandardWrapper 类使用整型变量 nInstances 来保存这个数值：

```
private int nInstances = 0;
```

下面是 allocate() 方法的第二部分：

```
synchronized (instancePool) {
    while (countAllocated >= nInstances) {
        // Allocate a new instance if possible, or else wait
        if (nInstances < maxInstances) {
            try {
                instancePool.push(loadServlet());
                nInstances++;
            }
            catch (ServletException e) {
                throw e;
            }
            catch (Throwable e) {
                throw new ServletException
                    (sm.getString("standardWrapper.allocate"), e);
            }
        }
        else {
            try {
                instancePool.wait();
            }
            catch (InterruptedException e) {
                ;
            }
        }
    }
    if (debug >= 2)
        log(" Returning allocated STM instance");
    countAllocated++;
}
```

```

        return (Servlet) instancePool.pop();
    }

```

上面的代码会使用一个 while 循环等待，直到变量 `nInstances` 的值小于等于变量 `countAllocated` 的值。在循环体内部，`allocate()` 方法会检查 `nInstances` 的值，如果它比变量 `maxInstances` 的值小，`allocate()` 方法会调用 `loadServlet()` 方法，并将新创建的 STM servlet 实例压入对象池中，并将变量 `nInstances` 的值加 1。如果变量 `nInstances` 的值大于等于变量 `maxInstance` 的值，它会通过调用变量 `instancePool` 栈的 `wait()` 方法进入等待状态，直到某个 STM servlet 实例被放回到栈中。

### 11.3.2 载入 servlet 类

`StandardWrapper` 类实现 `Wrapper` 接口的 `load()` 方法，`load()` 方法调用 `loadServlet()` 方法载入某个 servlet 类，并调用其 `init()` 方法，此时要传入一个 `javax.servlet.ServletConfig` 实例作为参数。下面来看一下 `loadServlet()` 方法是如何工作的。

`loadServlet()` 方法首先会检查当前的 `StandardWrapper` 类是否表示的是一个 STM servlet 类，若不是，且变量 `instance` 不为 `null`（表示以前已经载入过这个 servlet 类），它就直接返回该实例：

```

// Nothing to do if we already have an instance or an instance pool
if (!singleThreadModel && (instance != null))
    return instance;

```

若 `instance` 为 `null`，或该 servlet 实例是一个 STM servlet，则执行后续的方法。

首先，它获取 `System.out` 和 `System.err` 的输出，便于它使用 `javax.servlet.ServletContext` 的 `log()` 方法记录日志消息：

```

PrintStream out = System.out;
SystemLogHandler.startCapture();

```

然后，它定义类型为 `javax.servlet.Servlet` 名为 `servlet` 的变量，变量 `servlet` 表示已载入的 servlet 类的实例，该实例将会由 `loadServlet()` 方法返回：

```

Servlet servlet = null;

```

`loadServlet()` 方法负责载入该 servlet 类，原先类名会保存在类变量 `ServletClass` 中，现在 `loadServlet()` 方法要将变量名写入到字符串变量 `actualClass` 中：

```

String actualClass = servletClass;

```

但是，由于 Catalina 也是一个 JSP 容器，因此 `loadServlet()` 方法必须检查请求的 servlet 是不是一个 JSP 页面。若是，`loadServlet()` 方法需要获取代表该 JSP 页面的实际的 servlet 类：

```

if ((actualClass == null) && (jspFile != null)) {
    Wrapper jspWrapper = (Wrapper)
        ((Context) getParent()).findChild(Constants.JSP_SERVLET_NAME);
    if (jspWrapper != null)
        actualClass = jspWrapper.getServletClass();
}

```

如果找不到该 JSP 页面的 servlet 类，则会使用变量 `servletclass` 的值。但是，若没有调用 `StandardWrapper` 类的 `setServletClass()` 方法设置 `servletClass` 的值，则会抛出异常，并停止执行后续的方法：

```
// Complain if no servlet class has been specified
if (actualClass == null) {
    unavailable(null);
    throw new ServletException
        (sm.getString("standardWrapper.notClass", getName()));
}
```

这时，要载入的 servlet 类名已经解析完，`loadServlet()` 方法会获取载入器。若找不到载入器，则它抛出异常，方法终止：

```
// Acquire an instance of the class loader to be used
Loader loader = getLoader();
if (loader == null) {
    unavailable(null);
    throw new ServletException
        (sm.getString("standardWrapper.missingLoader", getName()));
}
```

若可以找到载入器，则 `loadServlet` 方法调用载入器的 `getClassLoader()` 方法获取一个 `ClassLoader`：

```
ClassLoader classLoader = loader.getClassLoader();
```

在 `org.apache.catalina` 包下，`Catalina` 提供了一些用于访问 servlet 容器内部数据的专用 servlet 类。如果某个 servlet 类是这种专用的 servlet，即，若 `isContainerProvidedServlet()` 方法返回 `true`，则变量 `classLoader` 的赋值为另一种 `ClassLoader` 实例，如此一来，这个 servlet 类的实例就可以访问 `Catalina` 的内部数据了：

```
// Special case class loader for a container provided servlet
if (isContainerProvidedServlet(actualClass)) {
    classLoader = this.getClass().getClassLoader();
    log(sm.getString
        ("standardWrapper.containerServlet", getName()));
}
```

有了类载入器和准备载入的 servlet 类名后，`loadServlet()` 方法就可以载入 servlet 类了：

```
// Load the specified servlet class from the appropriate class
// loader
Class classClass = null;
try {
    if (classLoader != null) {
        System.out.println("Using classLoader.loadClass");
        classClass = classLoader.loadClass(actualClass);
    }
    else {
        System.out.println("Using forName");
        classClass = Class.forName(actualClass);
    }
}
catch (ClassNotFoundException e) {
    unavailable(null);
    throw new ServletException
        (sm.getString("standardWrapper.missingClass", actualClass), e);
}
```



```

if (classClass == null) {
    unavailable(null);
    throw new ServletException
        (sm.getString("standardWrapper.missingClass", actualClass));
}

```

然后, 实例化该 servlet 类:

```

// Instantiate and initialize an instance of the servlet class
// itself
try {
    servlet = (Servlet) classClass.newInstance();
}
catch (ClassCastException e) {
    unavailable(null);
    // Restore the context ClassLoader
    throw new ServletException
        (sm.getString("standardWrapper.notServlet", actualClass), e);
}
catch (Throwable e) {
    unavailable(null);
    // Restore the context ClassLoader
    throw new ServletException
        (sm.getString("standardWrapper.instantiate", actualClass), e);
}

```

但是, 在 `loadMethod()` 方法实例化这个 servlet 类之前, 它会调用 `isServletAllowed()` 方法检查该 servlet 类是否允许载入:

```

// Check if loading the servlet in this web application should be
// allowed
if (!isServletAllowed(servlet)) {
    throw new SecurityException
        (sm.getString("standardWrapper.privilegedServlet",
            actualClass));
}

```

若通过了安全检查, 它会继续检查该 servlet 类是否是一个 `ContainerServlet` 类型的 servlet。实现了 `org.apache.catalina.ContainerServlet` 接口的 servlet 可以访问 Catalina 的内部功能。若该 servlet 类是一个 `ContainerServlet`, `loadServlet()` 方法会调用 `ContainerServlet` 接口的 `setWrapper()` 方法, 传入这个 `StandardWrapper` 实例:

```

// Special handling for ContainerServlet instances
if ((servlet instanceof ContainerServlet) &&
    isContainerProvidedServlet(actualClass)) {
    ((ContainerServlet) servlet).setWrapper(this);
}

```

接下来, `loadServlet()` 方法触发 `BEFORE_INIT_EVENT` 事件, 调用 servlet 实例的 `init()` 方法:

```

try {
    instanceSupport.fireInstanceEvent(
        InstanceEvent.BEFORE_INIT_EVENT, servlet);
    servlet.init(facade);
}

```

注意, `init()` 方法传入了引用 `javax.servlet.ServletConfig` 对象的一个外观变量。11.3.3 节将会对如何创建 `ServletConfig` 对象进行介绍。

若变量 `loadOnStartup` 的值大于 0, 而且被请求的 servlet 类实际上是一个 JSP 页面, 则也调

用 servlet 的 service() 方法:

```
// Invoke jspInit on JSP pages
if ((loadOnStartup > 0) && (jspFile != null)) {
    // Invoking jspinit
    HttpRequestBase req = new HttpRequestBase();
    HttpResponseBase res = new HttpResponseBase();
    req.setServletPath(jspFile);
    req.setQueryString("jsp_precompile=true");
    servlet.service(req, res);
}
```

接下来, loadServlet() 方法会触发 AFTER\_INIT\_EVENT 事件:

```
instanceSupport.fireInstanceEvent (InstanceEvent.AFTER_INIT_EVENT, servlet);
```

若 StandardWrapper 对象表示的 servlet 类是一个 STM servlet, 则将该 servlet 实例添加到 servlet 实例池中。因此会先判断变量 instancePool 的值是否为 null, 若是, 则要给它赋值一个 Stack 对象:

```
// Register our newly initialized instance
singleThreadModel = servlet instanceof SingleThreadModel;
if (singleThreadModel) {
    if (instancePool == null)
        instancePool = new Stack();
}
fireContainerEvent("load", this);
}
```

在 finally 代码块中, loadServlet() 方法停止捕获 System.out 和 System.err 对象, 记录在载入 ServletContext 的 log() 方法的过程中产生的日志消息:

```
finally {
    String log = SystemLogHandler.stopCapture();
    if (log != null && log.length() > 0) {
        if (getServletContext() != null) {
            getServletContext().log(log);
        }
        else {
            out.println(log);
        }
    }
}
```

最后, loadServlet() 方法返回已载入的 servlet 实例:

```
return servlet;
```

### 11.3.3 ServletConfig 对象

StandardWrapper 类的 loadServlet() 方法在载入 servlet 类后, 会调用该 servlet 实例的 init() 方法。init() 方法需要传入一个 javax.servlet.ServletConfig 实例作为参数。也许你很想知道 StandardWrapper 对象是如何获取 servletConfig 对象的。

答案就在 StandardWrapper 类本身中。该类不仅实现了 Wrapper 接口, 还实现了 javax.servlet.ServletConfig 接口。

ServletConfig 接口有以下 4 个方法, getServletContext()、getServletName()、getInitParameter()

和 `getInitParameterNames()`。下面来看一下这些方法在 `StandardWrapper` 类中这些方法的具体实现。

**注意** `StandardWrapper` 类并不将自身传递给 `servlet` 实例的 `init()` 方法。相反，它会在一个 `StandardWrapperFacade` 实例中包装自身，将其大部分的公共方法对 `servlet` 程序员隐藏起来。详细内容参见 11.4 节。

### 1. `getServletContext()` 方法

该方法的签名如下：

```
public ServletContext getServletContext()
```

`StandardWrapper` 实例肯定是 `StandardContext` 实例的子容器，也就是说，`StandardWrapper` 实例的父容器是 `StandardContext` 实例。对于 `StandardContext` 对象，可以调用 `StandardContext` 对象的 `getServletContext()` 方法来获得一个 `ServletContext` 实例。下面是 `StandardWrapper` 类中 `getServletContext()` 方法的实现：

```
public ServletContext getServletContext() {
    if (parent == null)
        return (null);
    else if (!(parent instanceof Context))
        return (null);
    else
        return (((Context) parent).getServletContext());
}
```

**注意** 正如上面的代码所展示的那样，无法单独使用一个 `Wrapper` 实例来表示一个 `servlet` 类的定义。`Wrapper` 实例必须驻留在某个 `Context` 容器中，这样，当调用其父容器的 `getServletContext()` 方法时才能返回 `ServletContext` 类的一个实例。

### 2. `getServletName` 方法

该方法返回 `servlet` 类的名字，该方法的签名如下：

```
public java.lang.String getServletName()
```

下面是 `StandardWrapper` 类中 `getServletName()` 方法的实现：

```
public String getServletName() {
    return (getName());
}
```

该方法仅仅是简单地调用了 `ContainerBase` 类（`StandardWrapper` 的父类）的 `getName()` 方法，`ContainerBase` 类中 `getName()` 方法的实现如下：

```
public String getName() {
    return (name);
}
```

可以通过调用 `setName()` 方法设置变量 `name` 的值。回忆一下，通过传递 `Servlet` 的名称可以调用 `StandardWrapper` 实例的 `setName()` 方法。

### 3. `getInitParameter()` 方法

该方法返回指定初始参数的值。该方法的签名如下：



```
public java.lang.String getInitParameter(java.lang.String name)
```

在 StandardWrapper 类中, 初始化参数存储在一个 HashMap 类型的名为 parameters 的变量中:

```
private HashMap parameters = new HashMap();
```

调用 StandardWrapper 类的 addInitParameter() 方法, 并传入参数的名字和对应的值来填充变量 parameters 的值:

```
public void addInitParameter(String name, String value) {  
    synchronized (parameters) {  
        parameters.put(name, value);  
    }  
    fireContainerEvent("addInitParameter", name);  
}
```

下面是 StandardWrapper 类中 getInitParameter() 方法的实现:

```
public String getInitParameter(String name) {  
    return (findInitParameter(name));  
}
```

其中 findInitParameter() 方法接受一个指定的初始化参数名的字符串变量, 调用 HashMap 变量 parameters 的 get() 方法获取初始化参数的值。下面是 findInitParameter() 方法的实现。

```
public String findInitParameter(String name) {  
    synchronized (parameters) {  
        return ((String) parameters.get(name));  
    }  
}
```

#### 4. getInitParameterNames() 方法

该方法返回所有初始化参数的名字的集合, 实际上是一个枚举类型 java.util Enumeration 的实例。下面是该方法的签名:

```
public java.util Enumeration getInitParameterNames()
```

在 StandardWrapper 类中, getInitParameterNames() 方法的具体实现如下:

```
public Enumeration getInitParameterNames() {  
    synchronized (parameters) {  
        return (new Enumerator(parameters.keySet()));  
    }  
}
```

其中, Enumerator 类实现了 java.util Enumeration 接口, 并且位于 org.apache.catalina.util 包下。

### 11.3.4 servlet 容器的父子关系

Wrapper 实例代表一个 servlet 实例, 因此, Wrapper 实例不能再有子容器, 不应该再调用其 addChild() 方法, 否则抛出 java.lang.IllegalStateException 异常。下面是 StandardWrapper 类中 addChild() 方法的实现:

```
public void addChild(Container child) {  
    throw new IllegalStateException  
        (sm.getString("standardWrapper.notChild"));  
}
```

Wrapper 的父容器只能是 Context 类的实现，若是在调用 Wrapper 实例的 setParent() 方法时传入了一个非 Context 类型的容器，则会抛出 java.lang.IllegalArgumentException 异常：

```
public void setParent(Container container) {
    if ((container != null) && !(container instanceof Context))
        throw new IllegalArgumentException
            (sm.getString("standardWrapper.notContext"));
    super.setParent(container);
}
```

## 11.4 StandardWrapperFacade 类

StandardWrapper 实例会调用它所载入的 servlet 类的实例的 init() 方法。init() 方法需要一个 javax.servlet.ServletConfig 实例，而 StandardWrapper 类本身实现了 javax.servlet.ServletConfig 接口，所以，理论上 StandardWrapper 对象可以将自己传入 init() 方法。但是，StandardWrapper 需要将其大部分公共方法对 servlet 程序员隐藏起来。为了实现在这个目的，StandardWrapper 类将自身实例包装成 StandardWrapperFacade 类的一个实例。图 11-2 展示了 StandardWrapper 类与 StandardWrapperFacade 类的关系，它们都实现了 javax.servlet.ServletConfig 接口。

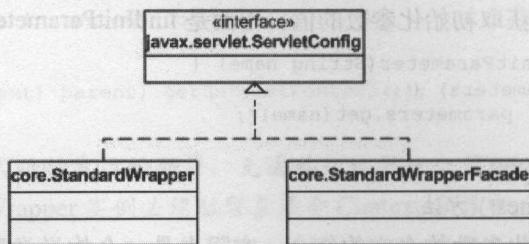


图 11-2 StandardWrapper 类与 StandardWrapperFacade 类之间的关系示意图

StandardWrapper 类使用下面的代码创建 StandardWrapperFacade 类的一个实例，需要将自身实例作为参数传入 StandardWrapperFacade 的构造函数：

```
private StandardWrapperFacade facade = new StandardWrapperFacade(this);
```

StandardWrapperFacade 类提供了一个 servletConfig 类型的类级别的变量 config：

```
private ServletConfig config = null;
```

当在 StandardWrapper 对象内创建 StandardWrapperFacade 实例时，StandardWrapperFacade 类的构造函数需要传入 StandardWrapper 对象，并为变量 config 赋值：

```
public StandardWrapperFacade(StandardWrapper config) {
    super();
    this.config = (ServletConfig) config;
}
```

因此，当创建 StandardWrapper 对象调用 servlet 实例的 init() 方法时，它会传入 StandardWrapperFacade 类的一个实例。这样，在 servlet 实例内调用 ServletConfig 类的 getServletName()、getInitParameter() 和 getInitParameterNames() 方法会直接传递给 StandardWrapper 类实现的相应方法：

```

public String getServletName() {
    return config.getServletName();
}
public String getInitParameter(String name) {
    return config.getInitParameter(name);
}
public Enumeration getInitParameterNames() {
    return config.getInitParameterNames();
}

```

对 `getServletContext()` 方法的调用会稍复杂一点:

```

public ServletContext getServletContext() {
    ServletContext theContext = config.getServletContext();
    if ((theContext != null) && (theContext instanceof
ApplicationContext))
        theContext = ((ApplicationContext) theContext).getFacade();
    return (theContext);
}

```

该方法会调用 `StandardWrapper` 类的 `getServletContext()` 方法,但是它返回的是 `ServletContext` 类的外观类的实例,而不是 `ServletContext` 对象本身。

## 11.5 StandardWrapperValve 类

`StandardWrapperValve` 类是 `StandardWrapper` 实例中的基础阀,要完成两个操作:

- 1) 执行与该 servlet 实例关联的全部过滤器;
- 2) 调用 servlet 实例的 `service()` 方法。

为完成上述任务,在 `StandardWrapperValve` 的 `invoke()` 方法实现中会执行以下几个操作:

- 1) 调用 `StandardWrapper` 实例的 `allocate()` 方法获取该 `StandardWrapper` 实例所表示的 servlet 实例;
- 2) 调用私有方法 `createFilterChain()`, 创建过滤器链;
- 3) 调用过滤器链的 `doFilter()` 方法,其中包括调用 servlet 实例的 `service()` 方法;
- 4) 释放过滤器链;
- 5) 调用 `Wrapper` 实例的 `deallocate()` 方法;
- 6) 若该 servlet 类再也不会被使用到,则调用 `Wrapper` 实例的 `unload()` 方法。

下面是 `invoke()` 方法的部分代码:

```

// Allocate a servlet instance to process this request
try {
    if (!unavailable) {
        servlet = wrapper.allocate();
    }
}
...
// Acknowledge the request
try {
    response.sendAcknowledgement();
}
...
// Create the filter chain for this request
ApplicationFilterChain filterChain = createFilterChain(request,
    servlet);

```



```

// Call the filter chain for this request
// This also calls the servlet's service() method
try {
    String jspFile = wrapper.getJspFile();
    if (jspFile != null)
        sreq.setAttribute(Globals.JSP_FILE_ATTR, jspFile);
    else
        sreq.removeAttribute(Globals.JSP_FILE_ATTR);
    if ((servlet != null) && (filterChain != null)) {
        filterChain.doFilter(sreq, sres);
    }
    sreq.removeAttribute(Globals.JSP_FILE_ATTR);
}
...
// Release the filter chain (if any) for this request
try {
    if (filterChain != null)
        filterChain.release();
}
...
// Deallocate the allocated servlet instance
try {
    if (servlet != null) {
        wrapper.deallocate(servlet);
    }
}
...
// If this servlet has been marked permanently unavailable,
// unload it and release this instance
try {
    if ((servlet != null) && (wrapper.getAvailable() ==
        Long.MAX_VALUE)) {
        wrapper.unload();
    }
}
...

```

其中最重要的是对 `createFilterChain()` 方法和过滤器链的 `doFilter()` 方法的调用。`createFilterChain()` 方法创建一个 `ApplicationFilterChain` 实例，并将所有需要应用到该 `Wrapper` 实例所代表的 `servlet` 实例的过滤器添加到其中。11.8 节将会对 `ApplicationFilterChain` 类进行介绍。但是，为了更好地理解 `ApplicationFilterChain` 类的作用，首先要了解 `FilterDef` 类和 `ApplicationFilterConfig` 类的工作机制。这两个类会在下面两节中介绍。

## 11.6 FilterDef 类

`org.apache.catalina.deploy.FilterDef` 类表示一个过滤器定义，正如在部署描述器中定义的 `filter` 元素那样。代码清单 11-1 给出了 `FilterDef` 类的定义。

代码清单 11-1 `FilterDef` 类的定义

```

package org.apache.catalina.deploy;

import java.util.HashMap;
import java.util.Map;

public final class FilterDef {
    /**

```

```

    * The description of this filter.
    */
    private String description = null;
    public String getDescription() {
        return (this.description);
    }
    public void setDescription(String description) {
        this.description = description;
    }

    /**
     * The display name of this filter.
     */
    private String displayName = null;
    public String getDisplayName() {
        return (this.displayName);
    }
    public void setDisplayName(String displayName) {
        this.displayName = displayName;
    }

    /**
     * The fully qualified name of the Java class that implements this
     * filter.
     */
    private String filterClass = null;
    public String getFilterClass() {
        return (this.filterClass);
    }
    public void setFilterClass(String filterClass) {
        this.filterClass = filterClass;
    }

    /**
     * The name of this filter, which must be unique among the filters
     * defined for a particular web application.
     */
    private String filterName = null;
    public String getFilterName() {
        return (this.filterName);
    }
    public void setFilterName(String filterName) {
        this.filterName = filterName;
    }

    /**
     * The large icon associated with this filter.
     */
    private String largeIcon = null;
    public String getLargeIcon() {
        return (this.largeIcon);
    }
    public void setLargeIcon(String largeIcon) {
        this.largeIcon = largeIcon;
    }

    /**
     * The set of initialization parameters for this filter, keyed by
     * parameter name.
     */
    private Map parameters = new HashMap();
    public Map getParameterMap() {

```

```

    return (this.parameters);
}

/**
 * The small icon associated with this filter.
 */
private String smallIcon = null;
public String getSmallIcon() {
    return (this.smallIcon);
}

public void setSmallIcon(String smallIcon) {
    this.smallIcon = smallIcon;
}

public void addInitParameter(String name, String value) {
    parameters.put(name, value);
}

/**
 * Render a String representation of this object.
 */
public String toString() {
    StringBuffer sb = new StringBuffer("FilterDef[");
    sb.append("filterName=");
    sb.append(this.filterName);
    sb.append(", filterClass=");
    sb.append(this.filterClass);
    sb.append("]");
    return (sb.toString());
}
}

```

FilterDef 类中的每个属性表示在定义 filter 元素时声明的子元素。其中 Map 类型的变量 parameters 存储了初始化过滤器时所需要的所有参数。addInitParameter() 方法用于向 parameters 中添加新的 name/value 形式的参数名和对应的值。

## 11.7 ApplicationFilterConfig 类

org.apache.catalina.core.ApplicationFilterConfig 类实现 javax.servlet.FilterConfig 接口。ApplicationFilterConfig 类用于管理 Web 应用程序第 1 次启动时创建的所有的过滤器实例。

可以通过把一个 org.apache.catalina.Context 对象和一个 FilterDef 对象传递给 ApplicationFilterConfig 类的构造函数来创建一个 ApplicationFilterConfig 对象：

```

public ApplicationFilterConfig(Context context, FilterDef filterDef)
    throws ClassCastException, ClassNotFoundException,
        IllegalAccessException, InstantiationException, ServletException

```

其中 Context 对象表示一个 Web 应用程序，FilterDef 对象表示一个过滤器的定义。ApplicationFilterConfig 类的 getFilter() 方法会返回一个 javax.servlet.Filter 对象。该方法负责载入并实例化一个过滤器类：

```

Filter getFilter() throws ClassCastException, ClassNotFoundException,
    IllegalAccessException, InstantiationException, ServletException {

    // Return the existing filter instance, if any
    if (this.filter != null)
        return (this.filter);
}

```



```
// Identify the class loader we will be using
String filterClass = filterDef.getFilterClass();
ClassLoader classLoader = null;
if (filterClass.startsWith("org.apache.catalina."))
    classLoader = this.getClass().getClassLoader();
else
    classLoader = context.getLoader().getClassLoader();

ClassLoader oldCtxClassLoader =
    Thread.currentThread().getContextClassLoader();

// Instantiate a new instance of this filter and return it
Class clazz = classLoader.loadClass(filterClass);
this.filter = (Filter) clazz.newInstance();
filter.init(this);
return (this.filter);
}
```

## 11.8 ApplicationFilterChain 类

org.apache.catalina.core.ApplicationFilterChain 类实现 javax.servlet.FilterChain 接口，StandardWrapperValve 类的 invoke() 方法会创建 ApplicationFilterChain 类的一个实例，并调用其 doFilter() 方法。ApplicationFilterChain 类的 doFilter() 方法会调用过滤器链中第 1 个过滤器的 doFilter() 方法。Filter 接口的 doFilter() 方法的签名是：

```
public void doFilter(ServletRequest request, ServletResponse response,
    FilterChain chain) throws java.io.IOException, ServletException
```

ApplicationFilterChain 类的 doFilter() 方法会将 ApplicationFilterChain 类自身作为第 3 个参数传给过滤器的 doFilter() 方法。

在其 doFilter() 方法中，可以通过显式地调用 FilterChain 对象的 doFilter() 方法来调用另一个过滤器。下面的代码演示了过滤器中 doFilter() 方法的实现：

```
public void doFilter(ServletRequest request, ServletResponse response,
    FilterChain chain) throws IOException, ServletException {
    // do something here
    ...
    chain.doFilter(request, response);
}
```

正如你所看到的，在 doFilter() 方法的最后一行会调用 FilterChain 实例的 doFilter() 方法。如果某个过滤器是过滤器链中的最后一个过滤器，则会调用被请求的 servlet 类的 service() 方法。如果过滤器没有调用 chain.doFilter() 方法，则不会调用后面的过滤器。

## 11.9 应用程序

本章的应用程序包含两个类，分别是 ex11.pyrmont.core.SimpleContextConfig 和 ex11.pyrmont.startup.Bootstrap，其中 SimpleContextConfig 类与之前章节的应用程序中的副本相同。代码清单 11-2 给出了 Bootstrap 类的定义。

## 代码清单 11-2 Bootstrap 类的定义

```

package ex11.pyrmont.startup;
//use StandardWrapper
import ex11.pyrmont.core.SimpleContextConfig;
import org.apache.catalina.Connector;
import org.apache.catalina.Context;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleListener;
import org.apache.catalina.Loader;
import org.apache.catalina.Wrapper;
import org.apache.catalina.connector.http.HttpConnector;
import org.apache.catalina.core.StandardContext;
import org.apache.catalina.core.StandardWrapper;
import org.apache.catalina.loader.WebappLoader;

public final class Bootstrap {
    public static void main(String[] args) {
        System.setProperty("catalina.base",
            System.getProperty("user.dir"));
        Connector connector = new HttpConnector();
        Wrapper wrapper1 = new StandardWrapper();
        wrapper1.setName("Primitive");
        wrapper1.setServletClass("PrimitiveServlet");
        Wrapper wrapper2 = new StandardWrapper();
        wrapper2.setName("Modern");
        wrapper2.setServletClass("ModernServlet");

        Context context = new StandardContext();
        // StandardContext's start method adds a default mapper
        context.setPath("/myApp");
        context.setDocBase("myApp");
        LifecycleListener listener = new SimpleContextConfig();
        ((Lifecycle) context).addLifecycleListener(listener);
        context.addChild(wrapper1);
        context.addChild(wrapper2);
        // for simplicity, we don't add a valve, but you can add
        // valves to context or wrapper just as you did in Chapter 6
        Loader loader = new WebappLoader();
        context.setLoader(loader);
        // context.addServletMapping(pattern, name);
        context.addServletMapping("/Primitive", "Primitive");
        context.addServletMapping("/Modern", "Modern");
        // add ContextConfig. This listener is important because it
        // configures StandardContext (sets configured to true), otherwise
        // StandardContext won't start
        connector.setContainer(context);
        try {
            connector.initialize();
            ((Lifecycle) connector).start();
            ((Lifecycle) context).start();
            // make the application wait until we press a key.
            System.in.read();
            ((Lifecycle) context).stop();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Bootstrap 类会创建一个 StandardContext 实例，称其所表示的 Web 应用程序为“MyApp”。而对于 StandardContext 实例，Bootstrap 类为其添加了两个 StandardWrapper 实例，分别是 Primitive 和 Modern。

## 运行应用程序

要在 Windows 平台下运行应用程序，需要在工作目录中执行以下命令：

```
java -classpath ./lib/servlet.jar;./lib/commons-collections.jar;./  
ex11.pyrmont.startup.Bootstrap
```

而在 Linux 平台下，需要使用冒号来分隔不同库：

```
java -classpath ./lib/servlet.jar:./lib/commons-collections.jar:./  
ex11.pyrmont.startup.Bootstrap
```

要调用 PrimitiveServlet，需要在浏览器中输入以下 URL：

```
http://localhost:8080/Primitive
```

要调用 ModernServlet，需要在浏览器中输入以下 URL：

```
http://localhost:8080/Modern
```

## 11.10 小结

在本章中，你已经学习了 StandardWrapper 类的工作原理，该类是 Catalina 中 Wrapper 接口的标准实现。此外，本章还讨论了过滤器及与过滤器相关的一些类，结尾的应用程序展示了如何使用 StandardWrapper 类。



## 第 12 章

# StandardContext 类

正如你在前面章节中所看到的，Context 实例表示一个具体的 Web 应用程序，其中包含一个或多个 Wrapper 实例，每个 Wrapper 表示一个具体的 servlet 定义。但是，Context 容器还需要其他组件的支持，典型的如载入器和 Session 管理器。本章将对 org.apache.catalina.core.StandardContext 类的工作机制进行说明，该类是 Catalina 中 Context 接口的标准实现。

我们首先会回顾一下 StandardContext 类的实例化和配置，然后讨论与 StandardContext 类相关的 StandardContextMapper 类（存在于 Tomcat 4 中）和 ContextConfig 类。接下来，学习对于引入的每个 HTTP 请求的方法调用序列。然后，重新讨论一下 StandardContext 类的几个重要的属性。最后，本章最后一节将会介绍 Tomcat 5 中的 backgroundProcess() 方法。

**注意** 本章并没有相应的应用程序，StandardContext 类的内容与前面章节中的相同。

### 12.1 StandardContext 的配置

在创建了 StandardContext 实例后，必须调用其 start() 方法来为引入的每个 HTTP 请求提供服务。但是，由于某种原因，StandardContext 对象可能会启动失败。这时 StandardContext 对象的 available 属性会被设置为 false，available 属性表明 StandardContext 对象是否可用。

若要是 start() 方法正确执行，则表明 StandardContext 对象配置正确。在 Tomcat 的实际部署中，配置 StandardContext 对象需要一系列操作。正确设置后，StandardContext 对象才能读取并解析默认的 web.xml 文件，该文件位于 %CATALINA\_HOME%/conf 目录下，该文件的内容会应用到所有部署到 Tomcat 中的应用程序中。这也保证了 StandardContext 实例可以处理应用程序级的 web.xml 文件。此外，还会配置验证器阀和许可阀。

**注意** 有关于 StandardContext 配置的更多详细内容请参见第 15 章。

StandardContext 类的 configured 属性是一个布尔变量，表明 StandardContext 实例是否正确设置。StandardContext 类使用一个事件监听器作为其配置器。当调用 StandardContext 实例的 start() 方法时，其中要做的一件事是，触发一个生命周期事件。该事件调用监听器，对 StandardContext 实例进行配置。若配置成功，监听器会将 configured 属性设置为 true。否则，StandardContext 实例会拒绝启动，也就无法为 HTTP 请求提供服务。

在第 11 章中，你已经学会了如何将生命周期监听器的实现添加到 StandardContext 实例中。它的类型是 ch11.pyrmont.core.SimpleContextConfig，它仅仅是将 StandardContext 实例的 configured 属性设置为 true，不执行其他的操作，只是使 StandardContext 实例认为它已经正确

地配置过了。在 Tomcat 的实际部署中，负责配置 StandardContext 实例的生命周期监听器是 org.apache.catalina.startup.ContextConfig 类的实例，该类将在第 15 章详细讨论。

既然你已经了解了配置 StandardContext 实例的重要性，我们就要深入了解 StandardContext 类的工作原理，先从它的构造函数开始。

### 12.1.1 StandardContext 类的构造函数

下面是 StandardContext 类的构造函数的实现：

```
public StandardContext() {
    super();
    pipeline.setBasic(new StandardContextValve());
    namingResources.setContainer(this);
}
```

构造函数中最重要的事情是为 StandardContext 实例的管道对象设置基础阀，其类型是 org.apache.catalina.core.StandardContextValve 类。该基础阀会处理从连接器中接收到的每个 HTTP 请求。

### 12.1.2 启动 StandardContext 实例

start() 方法会初始化 StandardContext 对象，用生命周期监听器配置 StandardContext 实例。当配置成功后，监听器会将其 configured 属性置为 true。最后，start() 方法会将 available 属性设置为 true 或者 false，true 表明 StandardContext 对象设置正确，与其相关联的子容器和组件都正确启动，因此，StandardContext 实例可以准备为引入的 HTTP 请求提供服务了。若期间发生了错误，则 available 的值会被置为 false。

StandardContext 使用一个初始化为 false 的布尔型变量 configured 来表明 StandardContext 对象是否正确配置。如果生命周期监听器成功执行了其配置 StandardContext 实例的任务，它就会把 StandardContext 的 configured 属性设置为 true。在 start() 方法的结尾，StandardContext 实例会检查 configured 变量的值，若 configured 为 true，则 StandardContext 启动成功；否则，调用 stop() 方法，关闭在 start() 方法已经启动的所有组件。

代码清单 12-1 给出了 Tomcat 4 中 StandardContext 类的 start() 方法的实现。

代码清单 12-1 Tomcat 4 中 StandardContext 类的 start() 方法的实现

```
public synchronized void start() throws LifecycleException {
    if (started)
        throw new LifecycleException(
            (sm.getString("containerBase.alreadyStarted", logName())));
    if (debug >= 1)
        log("Starting");
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);

    if (debug >= 1)
        log("Processing start(), current available=" + getAvailable());
    setAvailable(false);
    setConfigured(false);
    boolean ok = true;

    // Add missing components as necessary
```

```

if (getResources() == null) { // (1) Required by Loader
    if (debug >= 1)
        log("Configuring default Resources");
    try {
        if ((docBase != null) && (docBase.endsWith(".war")))
            setResources(new WARDirContext());
        else
            setResources(new FileDirContext());
    }
    catch (IllegalArgumentException e) {
        log("Error initializing resources: " + e.getMessage());
        ok = false;
    }
}

if (ok && (resources instanceof ProxyDirContext)) {
    DirContext dirContext =
        ((ProxyDirContext) resources).getDirContext();
    if ((dirContext != null)
        && (dirContext instanceof BaseDirContext)) {
        ((BaseDirContext) dirContext).setDocBase(getBasePath());
        ((BaseDirContext) dirContext).allocate();
    }
}

if (getLoader() == null) { // (2) Required by Manager
    if (getPrivileged()) {
        if (debug >= 1)
            log("Configuring privileged default Loader");
        setLoader(new WebappLoader(this.getClass().getClassLoader()));
    }
    else {
        if (debug >= 1)
            log("Configuring non-privileged default Loader");
        setLoader(new WebappLoader(getParentClassLoader()));
    }
}

if (getManager() == null) { // (3) After prerequisites
    if (debug >= 1)
        log("Configuring default Manager");
    setManager(new StandardManager());
}

// Initialize character set mapper
getCharsetMapper();

// Post work directory
postWorkDirectory();

// Reading the "catalina.useNaming" environment variable
String useNamingProperty = System.getProperty("catalina.useNaming");
if ((useNamingProperty != null)
    && (useNamingProperty.equals("false"))) {
    useNaming = false;
}

if (ok && isUseNaming()) {
    if (namingContextListener == null) {
        namingContextListener = new NamingContextListener();
        namingContextListener.setDebug(getDebug());
        namingContextListener.setName(getNamingContextName());
        addLifecycleListener(namingContextListener);
    }
}

```



```

// Binding thread
ClassLoader oldCCL = bindThread();

// Standard container startup
if (debug >= 1)
    log("Processing standard container startup");

if (ok) {
    try {
        addDefaultMapper(this.mapperClass);
        started = true;

        // Start our subordinate components, if any
        if ((loader != null) && (loader instanceof Lifecycle))
            ((Lifecycle) loader).start();
        if ((logger != null) && (logger instanceof Lifecycle))
            ((Lifecycle) logger).start();

        // Unbinding thread
        unbindThread(oldCCL);

        // Binding thread
        oldCCL = bindThread();

        if ((cluster != null) && (cluster instanceof Lifecycle))
            ((Lifecycle) cluster).start();
        if ((realm != null) && (realm instanceof Lifecycle))
            ((Lifecycle) realm).start();
        if ((resources != null) && (resources instanceof Lifecycle))
            ((Lifecycle) resources).start();

        // Start our Mappers, if any
        Mapper mappers[] = findMappers();
        for (int i = 0; i < mappers.length; i++) {
            if (mappers[i] instanceof Lifecycle)
                ((Lifecycle) mappers[i]).start();
        }

        // Start our child containers, if any
        Container children[] = findChildren();
        for (int i = 0; i < children.length; i++) {
            if (children[i] instanceof Lifecycle)
                ((Lifecycle) children[i]).start();
        }

        // Start the Valves in our pipeline (including the basic),
        // if any
        if (pipeline instanceof Lifecycle)
            ((Lifecycle) pipeline).start();

        // Notify our interested LifecycleListeners
        lifecycle.fireLifecycleEvent(START_EVENT, null);

        if ((manager != null) && (manager instanceof Lifecycle))
            ((Lifecycle) manager).start();
    }
    finally {
        // Unbinding thread
        unbindThread(oldCCL);
    }
}

if (!getConfigured())

```

```

    ok = false;

    // We put the resources into the servlet context
    if (ok)
        getServletContext().setAttribute
            (Globals.RESOURCES_ATTR, getResources());

    // Binding thread
    oldCCL = bindThread();

    // Create context attributes that will be required
    if (ok) {
        if (debug >= 1)
            log("Posting standard context attributes");
        postWelcomeFiles();
    }

    // Configure and call application event listeners and filters
    if (ok) {
        if (!listenerStart())
            ok = false;
    }
    if (ok) {
        if (!filterStart())
            ok = false;
    }

    // Load and initialize all "load on startup" servlets
    if (ok)
        loadOnStartup(findChildren());

    // Unbinding thread
    unbindThread(oldCCL);

    // Set available status depending upon startup success
    if (ok) {
        if (debug >= 1)
            log("Starting completed");
        setAvailable(true);
    }
    else {
        log(sm.getString("standardContext.startFailed"));
        try {
            stop();
        }
        catch (Throwable t) {
            log(sm.getString("standardContext.startCleanup"), t);
        }
        setAvailable(false);
    }

    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(AFTER_START_EVENT, null);
}

```

**注意** 在 Tomcat 5 中，start() 方法虽然与 Tomcat 4 中的类似，但它包含了一些与 JMX 相关的代码，但如果你不理解 JMX 的话，这些代码是没有作用的。JMX 的相关内容请参见第 20 章。如果你从未接触过 JMX，那么当你学习完本章内容后，希望你能阅读一下 Tomcat 5 中的 start() 方法的代码。

正如代码清单 12-1 所示, start() 方法需要完成以下工作:

- 1) 触发 BEFORE\_START 事件;
- 2) 将 availability 属性设置为 false;
- 3) 将 configured 属性设置为 false;
- 4) 配置资源;
- 5) 设置载入器;
- 6) 设置 Session 管理器;
- 7) 初始化字符集映射器;
- 8) 启动与该 Context 容器相关联的组件;
- 9) 启动子容器;
- 10) 启动管道对象;
- 11) 启动 Session 管理器;
- 12) 触发 START 事件, 在这里监听器 (ContextConfig 实例) 会执行一些配置操作 (具体内容将在第 15 章讨论), 若设置成功, ContextConfig 实例会将 StandardContext 实例的 configured 变量设置为 true;
- 13) 检查 configured 属性的值, 若为 true, 则调用 postWelcomePages() 方法, 载入那些需要在启动时就载入的子容器, 即 Wrapper 实例, 将 availability 属性设置为 true。若 configured 变量为 false, 则调用 stop() 方法;
- 14) 触发 AFTER\_START 事件。

### 12.1.3 invoke() 方法

在 Tomcat 4 中, StandardContext 类的 invoke 方法由与其相关联的连接器调用, 或者当 StandardContext 实例是 Host 容器的一个子容器时, 由 Host 实例的 invoke() 方法调用。StandardContext 类的 invoke() 方法首先会检查应用程序是否正在重载过程中, 若是, 则等待应用程序重载完成。然后, 它调用其父类 ContainerBase 的 invoke() 方法。代码清单 12-2 展示了 StandardContext 类的 invoke() 方法的实现。

代码清单 12-2 StandardContext 类的 invoke() 方法的实现

```
public void invoke(Request request, Response response)
    throws IOException, ServletException {

    // Wait if we are reloading
    while (getPaused()) {
        try {
            Thread.sleep(1000);
        }
        catch (InterruptedException e) {
            ;
        }
    }
    // Normal request processing
    if (swallowOutput) {
        try {
```



```

        SystemLogHandler.startCapture();
        super.invoke(request, response);
    }
    finally {
        String log = SystemLogHandler.stopCapture();
        if (log != null && log.length() > 0) {
            log(log);
        }
    }
}
else {
    super.invoke(request, response);
}
}
}

```

其中, `getPaused()` 方法返回 `paused` 属性的值, 当 `paused` 的值为 `true` 时, 表明应用程序正在重载。应用程序的重载将在 12.3 节中介绍。

在 Tomcat 5 中, `StandardContext` 类并没有提供 `invoke()` 方法的实现, 因此会执行 `ContainerBase` 类的 `invoke()` 方法。检查应用程序是否正在重载的工作移到了 `StandardContextValve` 类的 `invoke()` 方法中。

## 12.2 StandardContextMapper 类

对于每个引入的 HTTP 请求, 都会调用 `StandardContext` 实例的管道对象的基础阀的 `invoke()` 方法来处理。`StandardContext` 实例的基础阀是 `org.apache.catalina.core.StandardContextValve` 类的实例。`StandardContextValve` 类的 `invoke()` 方法要做的第一件事是获取一个要处理 HTTP 请求的 `Wrapper` 实例。

在 Tomcat 4 中, `StandardContextValve` 实例在它包含的 `StandardContext` 中查找。`StandardContextValve` 实例使用 `StandardContext` 实例的映射器找到一个合适的 `Wrapper` 实例。获得了 `Wrapper` 实例后, 它就会调用 `Wrapper` 实例的 `invoke()` 方法。在深入挖掘 `StandardContextValve` 类的工作原理之前, 本节先要介绍一些映射器组件。

`ContainerBase` 类是 `StandardContext` 类的父类, 前者定义 `addDefaultMapper()` 方法用来添加一个默认的映射器, 如下所示:

```

protected void addDefaultMapper(String mapperClass) {
    // Do we need a default Mapper?
    if (mapperClass == null)
        return;
    if (mappers.size() >= 1)
        return;

    // Instantiate and add a default Mapper
    try {
        Class clazz = Class.forName(mapperClass);
        Mapper mapper = (Mapper) clazz.newInstance();
        mapper.setProtocol("http");
        addMapper(mapper);
    }
    catch (Exception e) {
    }
}

```

```

log(sm.getString("containerBase.addDefaultMapper", mapperClass),
    e);
}
}

```

StandardContext 类在其 start() 方法中调用 addDefaultMapper() 方法，并传入变量 mapperClass 的值：

```

public synchronized void start() throws LifecycleException {
    ...
    if (ok) {
        try {
            addDefaultMapper(this.mapperClass);
        }
        ...
    }
}

```

StandardContext 定义的 mapperClass 变量如下：

```

private String mapperClass =
"org.apache.catalina.core.StandardContextMapper";

```

必须要调用映射器的 setContainer() 方法，通过传入一个容器的实例，将映射器和容器相关联。在 Catalina 中，org.apache.catalina.Mapper 接口的实现类是 org.apache.catalina.core.StandardContextMapper 类。StandardContextMapper 实例只能与 Context 级容器相关联，如其 setContainer() 方法所示：

```

public void setContainer(Container container) {
    if (!(container instanceof StandardContext))
        throw new IllegalArgumentException
            (sm.getString("httpContextMapper.container"));
    context = (StandardContext) container;
}

```

映射器中最重要的是 map() 方法，该方法会返回用来处理 HTTP 请求的子容器，该方法的签名如下：

```

public Container map(Request request, boolean update)

```

在 StandardContextMapper 类中，map() 方法返回一个 Wrapper 实例，用于处理请求。若找不到合适的 Wrapper 实例，则该方法返回 null。

现在回到本节开始处的讨论，对于引入的每个 HTTP 请求，StandardContextValve 实例调用 Context 容器的 map() 方法，并传入一个 org.apache.catalina.Request 对象。map() 方法（实际上是定义在父类 ContainerBase 中的）会针对某个特定的协议调用 findMapper() 方法返回一个映射器对象，然后调用映射器对象的 map() 方法获取 Wrapper 实例：

```

// Select the Mapper we will use
Mapper mapper = findMapper(request.getRequest().getProtocol());
if (mapper == null)
    return (null);
// Use this Mapper to perform this mapping
return (mapper.map(request, update));

```

StandardContextMapper 类的 map() 方法会先标识出相对于 Context 的 URI：

```
// Identify the context-relative URI to be mapped
String contextPath =
    ((HttpServletRequest) request.getRequest()).getContextPath();
String requestURI = ((HttpRequest) request).getDecodedRequestURI();
String relativeURI = requestURI.substring(contextPath.length());
```

然后，它试图应用匹配规则找到一个适合的 Wrapper 实例：

```
// Apply the standard request URI mapping rules from the specification
Wrapper wrapper = null;
String servletPath = relativeURI;
String pathInfo = null;
String name = null;

// Rule 1 -- Exact Match
if (wrapper == null) {
    if (debug >= 2)
        context.log(" Trying exact match");
    if (!(relativeURI.equals("/")))
        name = context.findServletMapping(relativeURI);
    if (name != null)
        wrapper = (Wrapper) context.findChild(name);
    if (wrapper != null) {
        servletPath = relativeURI;
        pathInfo = null;
    }
}

// Rule 2 -- Prefix Match
if (wrapper == null) {
    if (debug >= 2)
        context.log(" Trying prefix match");
    servletPath = relativeURI;
    while (true) {
        name = context.findServletMapping(servletPath + "/*");
        if (name != null)
            wrapper = (Wrapper) context.findChild(name);
        if (wrapper != null) {
            pathInfo = relativeURI.substring(servletPath.length());
            if (pathInfo.length() == 0)
                pathInfo = null;
            break;
        }
        int slash = servletPath.lastIndexOf('/');
        if (slash < 0)
            break;
        servletPath = servletPath.substring(0, slash);
    }
}

// Rule 3 -- Extension Match
if (wrapper == null) {
    if (debug >= 2)
        context.log(" Trying extension match");
    int slash = relativeURI.lastIndexOf('/');
    if (slash >= 0) {
        String last = relativeURI.substring(slash);
        int period = last.lastIndexOf('.');
        if (period >= 0) {
            String pattern = "*" + last.substring(period);
```



```

        name = context.findServletMapping(pattern);
        if (name != null)
            wrapper = (Wrapper) context.findChild(name);
        if (wrapper != null) {
            servletPath = relativeURI;
            pathInfo = null;
        }
    }
}

// Rule 4 -- Default Match
if (wrapper == null) {
    if (debug >= 2)
        context.log(" Trying default match");
    name = context.findServletMapping("/");
    if (name != null)
        wrapper = (Wrapper) context.findChild(name);
    if (wrapper != null) {
        servletPath = relativeURI;
        pathInfo = null;
    }
}

```

你也许要问, Context 实例是如何得到这些信息用来映射 servlet 的呢? 回忆一下在第 11 章的 Bootstrap 类中使用下面的代码向 StandardContext 添加了两个 servlet 映射和两个 Wrapper 实例:

```

context.addServletMapping("/Primitive", "Primitive");
context.addServletMapping("/Modern", "Modern");
context.addChild(wrapper1);
context.addChild(wrapper2);

```

在 Tomcat 5 中, Mapper 接口及其相关类已经移除了。事实上, StandardContextValve 类的 invoke() 方法会从 request 对象中获取适合的 Wrapper 实例:

```
Wrapper wrapper = request.getWrapper();
```

该 Wrapper 实例指明了封装在 request 对象中的映射信息。

## 12.3 对重载的支持

StandardContext 类定义了 reloadable 属性来指明该应用程序是否启用了重载功能。当启用了重载功能后, 当 web.xml 文件发生变化或 WEB-INF/classes 目录下的其中一个文件被重新编译后, 应用程序会重载。

StandardContext 类是通过其载入器实现应用程序重载的。在 Tomcat 4 中, StandardContext 对象中的 WebappLoader 类实现了 Loader 接口, 并使用另一个线程检查 WEB-INF 目录中的所有类和 JAR 文件的时间戳。只需要调用其 setContainer() 方法将 WebappLoader 对象与 StandardContext 对象相关联就可以启动该检查线程。下面是 Tomcat 4 中 WebappLoader 类的 setContainer() 的实现代码:

```

public void setContainer(Container container) {
    // Deregister from the old Container (if any)
    if ((this.container != null) && (this.container instanceof Context))
        ((Context) this.container).removePropertyChangeListener(this);
    // Process this property change
}

```

```

Container oldContainer = this.container;
this.container = container;
support.firePropertyChange("container", oldContainer,
    this.container);
// Register with the new Container (if any)
if ((this.container!=null) && (this.container instanceof Context)) {
    setReloadable( ((Context) this.container).getReloadable() );
    ((Context) this.container).addChangeListener(this);
}
}

```

看一下上述代码最后一个 if 语句块。如果当前容器是 Context 容器，则调用 setReloadable() 方法。这说明，WebappLoader 实例的 reloadable 属性的值与 StandardContext 实例的 reloadable 属性的值相同。

下面是 WebappLoader 类的 setReloadable() 方法的实现代码：

```

public void setReloadable(boolean reloadable) {
    // Process this property change
    boolean oldReloadable = this.reloadable;
    this.reloadable = reloadable;
    support.firePropertyChange("reloadable",
        new Boolean(oldReloadable), new Boolean(this.reloadable));
    // Start or stop our background thread if required
    if (!started)
        return;
    if (!oldReloadable && this.reloadable)
        threadStart();
    else if (oldReloadable && !this.reloadable)
        threadStop();
}

```

若 reloadable 属性的值从 false 修改为 true，则会调用 threadStart() 方法；若 reloadable 属性的值从 true 修改为 false，则会调用 threadStop() 方法。threadStart() 方法会启动一个专用的线程来不断地检查 WEB-INF 目录下的类和 JAR 文件的时间戳，而 threadStop() 方法则会终止该线程。

在 Tomcat 5 中，为支持重载功能而进行的检查类的时间戳的工作改为由 backgroundProcess() 方法执行，12.4 节将介绍该方法。

## 12.4 backgroundProcess() 方法

Context 容器的运行需要其他组件的支持，例如载入器和 Session 管理器。通常来说，这些组件需要使用各自的线程执行一些后台处理任务。例如，为了支持自动重载，载入器需要使用一个线程周期性地检查 WEB-INF 目录下的所有类和 JAR 文件的时间戳；Session 管理器使用一个线程检查它所管理的 session 对象的过期时间。在 Tomcat 4 中，这些组件最终拥有各自的线程。

为了节省资源，在 Tomcat 5 中，使用了不同的方法。所有的后台处理共享同一个线程。若某个组件或 servlet 容器需要周期性地执行一个操作，只需要将代码写到其 backgroundProcess() 方法中即可。

这个共享线程在 ContainerBase 对象中创建。ContainerBase 类在其 start() 方法中（即，当该容器启动时）调用其 threadStart() 方法启动该后台线程：

```

protected void threadStart() {
    if (thread != null)
        return;
    if (backgroundProcessorDelay <= 0)
        return;
    threadDone = false;
    String threadName = "ContainerBackgroundProcessor[" + toString() +
        "]",
        thread = new Thread(new ContainerBackgroundProcessor(), threadName);
    thread.setDaemon(true);
    thread.start();
}

```

threadStart() 方法通过传入一个实现了 java.lang.Runnable 接口的 ContainerBackgroundProcessor 类的实例构造了一个新线程。代码清单 12-3 给出了 ContainerBackgroundProcessor 类的定义。

代码清单 12-3 ContainerBackgroundProcessor 类的定义

```

protected class ContainerBackgroundProcessor implements Runnable {
    public void run() {
        while (!threadDone) {
            try {
                Thread.sleep(backgroundProcessorDelay * 1000L);
            }
            catch (InterruptedException e) {
                ;
            }
            if (!threadDone) {
                Container parent = (Container) getMappingObject();
                ClassLoader cl =
                    Thread.currentThread().getContextClassLoader();
                if (parent.getLoader() != null) {
                    cl = parent.getLoader().getClassLoader();
                }
                processChildren(parent, cl);
            }
        }
    }

    protected void processChildren(Container container, ClassLoader cl) {
        try {
            if (container.getLoader() != null) {
                Thread.currentThread().setContextClassLoader
                    (container.getLoader().getClassLoader());
            }
            container.backgroundProcess();
        }
        catch (Throwable t) {
            log.error("Exception invoking periodic operation: ", t);
        }
        finally {
            Thread.currentThread().setContextClassLoader(cl);
        }
        Container[] children = container.findChildren();
        for (int i = 0; i < children.length; i++) {
            if (children[i].getBackgroundProcessorDelay() <= 0) {
                processChildren(children[i], cl);
            }
        }
    }
}

```



ContainerBackgroundProcessor 类实际上是 ContainerBase 类的内部类。在其 run() 方法中是一个 while 循环, 周期性地调用其 processChildren() 方法。而 processChildren() 方法会调用自身对象的 backgroundProcess() 方法和其每个子容器的 processChildren() 方法。通过实现 backgroundProcess() 方法, ContainerBase 类的子类可以使用一个专用线程来执行周期性任务, 例如检查类的时间戳或检查 session 对象的超时时间。代码清单 12-4 给出了 Tomcat 5 中 StandardContext 类的 backgroundProcess() 方法的实现。

代码清单 12-4 Tomcat 5 中 StandardContext 类的 backgroundProcess() 方法的实现

```
public void backgroundProcess() {
    if (!started)
        return;
    count = (count + 1) % managerChecksFrequency;
    if ((getManager() != null) && (count == 0)) {
        try {
            getManager().backgroundProcess();
        }
        catch (Exception x) {
            log.warn("Unable to perform background process on manager", x);
        }
    }
    if (getLoader() != null) {
        if (reloadable && (getLoader().modified())) {
            try {
                Thread.currentThread().setContextClassLoader
                    (StandardContext.class.getClassLoader());
                reload();
            }
            finally {
                if (getLoader() != null) {
                    Thread.currentThread().setContextClassLoader
                        (getLoader().getClassLoader());
                }
            }
        }
        if (getLoader() instanceof WebappLoader) {
            ((WebappLoader) getLoader()).closeJARs(false);
        }
    }
}
```

现在, 你应该已经了解了 StandardContext 实例是如何帮助与其相关联的载入器和 Session 管理器执行周期性任务的。

## 12.5 小结

在本章中, 你已经学习了 StandardContext 类及其相关类的工作原理, 包括如何配置 StandardContext 实例和 StandardContext 实例如何处理每个引入的 HTTP 请求。本章最后一节介绍了 Tomcat 5 中的 backgroundProcess() 方法的实现。

## 第 13 章

# Host 和 Engine

本章要讨论两个主题，分别是 Host 容器和 Engine 容器。如果你想在同一个 Tomcat 部署上运行多个 Context 容器的话，你就需要使用 Host 容器。理论上，当你只有一个 Context 实例时，不需要使用 Host 实例。在 org.apache.catalina.Context 接口的描述中有如下一段话：

Context 容器的父容器通常是 Host 容器，也有可能是其他实现，或者如果不必要，就可以不使用父容器。

但是在 Tomcat 的实际部署中，总是会使用一个 Host 容器。13.6 节将说明为什么一定要这样做。

Engine 容器表示 Catalina 的整个 servlet 引擎。如果使用了 Engine 容器，那么它总是处于容器层级的最顶层。添加到 Engine 容器中的子容器通常是 org.apache.catalina.Host 的实现或 org.apache.catalina.Context 的实现。默认情况下，Tomcat 会使用 Engine 容器的，并且有一个 Host 容器作为其子容器。

本章将会讨论一些与 Host 接口和 Engine 接口相关的类。首先介绍 Host 接口及其相关实现类 StandardHost 类、StandardHostMapper 类（只存在于 Tomcat 4 中）和 StandardHostValve 类。接下来使用一个应用程序来说明如何使用 Host 实例作为一个顶层 servlet 容器。然后介绍 Engine 接口与其相关实现类 StandardEngine 类和 StandardEngineValve 类，并使用另一个应用程序说明如何使用 Engine 实例作为顶层 servlet 容器。

### 13.1 Host 接口

Host 容器是 org.apache.catalina.Host 接口的实例。Host 接口继承自 Container 接口。代码清单 13-1 给出了 Host 接口的定义。

代码清单 13-1 Host 接口的定义

```
package org.apache.catalina;

public interface Host extends Container {
    public static final String ADD_ALIAS_EVENT = "addAlias";
    public static final String REMOVE_ALIAS_EVENT = "removeAlias";
    /**
     * Return the application root for this Host.
     * This can be an absolute
     * pathname, a relative pathname, or a URL.
     */
    public String getAppBase();

    /**
     * Set the application root for this Host. This can be an absolute
     * pathname, a relative pathname, or a URL.
     */
}
```

```

    * @param appBase The new application root
    */
    public void setAppBase(String appBase);

    /**
     * Return the value of the auto deploy flag.
     * If true, it indicates that
     * this host's child webapps should be discovered and automatically
     * deployed.
     */
    public boolean getAutoDeploy();

    /**
     * Set the auto deploy flag value for this host.
     *
     * @param autoDeploy The new auto deploy flag
     */
    public void setAutoDeploy(boolean autoDeploy);

    /**
     * Set the DefaultContext
     * for new web applications.
     *
     * @param defaultContext The new DefaultContext
     */
    public void addDefaultContext(DefaultContext defaultContext);

    /**
     * Retrieve the DefaultContext for new web applications.
     */
    public DefaultContext getDefaultContext();

    /**
     * Return the canonical, fully qualified, name of the virtual host
     * this Container represents.
     */
    public String getName();

    /**
     * Set the canonical, fully qualified, name of the virtual host
     * this Container represents.
     *
     * @param name Virtual host name
     *
     * @exception IllegalArgumentException if name is null
     */
    public void setName(String name);

    /**
     * Import the DefaultContext config into a web application context.
     *
     * @param context web application context to import default context
     */
    public void importDefaultContext(Context context);

    /**
     * Add an alias name that should be mapped to this same Host.
     *
     * @param alias The alias to be added
     */

```



```

public void addAlias(String alias);

/**
 * Return the set of alias names for this Host. If none are defined,
 * a zero length array is returned.
 */
public String[] findAliases();

/**
 * Return the Context that would be used to process the specified
 * host-relative request URI, if any; otherwise return
 * <code>null</code>.
 *
 * @param uri Request URI to be mapped
 */
public Context map(String uri);

/**
 * Remove the specified alias name from the aliases for this Host.
 * @param alias Alias name to be removed
 */
public void removeAlias(String alias);
}

```

Host 接口中比较重要的是 map() 方法，该方法返回一个用来处理引入的 HTTP 请求的 Context 容器的实例。该方法的具体实现是在 StandardHost 类中，将在 13.2 节中讨论。

## 13.2 StandardHost 类

在 Catalina 中 org.apache.catalina.core.StandardHost 类是 Host 接口的标准实现。该类继承自 org.apache.catalina.core.ContainerBase 类，实现了 Host 和 Deployer 接口。Deployer 接口将在第 17 章介绍。

与 StandardContext 类和 StandardWrapper 类相似，StandardHost 类的构造函数会将一个基础的实例添加到其管道对象中：

```

public StandardHost() {
    super();
    pipeline.setBasic(new StandardHostValve());
}

```

如你所见，基础阀是 org.apache.catalina.core.StandardHostValve 类的实例。

当调用其 start() 方法时，StandardHost 实例会添加两个阀，分别是 ErrorReportValve 类和 ErrorDispatcherValve 类的实例。这两个阀均位于 org.apache.catalina.valves 包下。代码清单 13-2 给出了 Tomcat 4 中 StandardHost 类的 start() 方法的实现。

代码清单 13-2 StandardHost 类中 start() 方法的实现

```

public synchronized void start() throws LifecycleException {
    // Set error report valve
    if ((errorReportValveClass != null)
        && (!errorReportValveClass.equals("")) ) {
        try {
            Valve valve =
                (Valve) Class.forName(errorReportValveClass).newInstance();

```

```

        addValve(valve);
    }
    catch (Throwable t) {
        log(sm.getString(
            "standardHost.invalidErrorReportValveClass",
            errorReportValveClass));
    }
}
// Set dispatcher valve
addValve(new ErrorDispatcherValve());
super.start();
}

```

**注意** 在 Tomcat 5 中, `start()` 方法与此类似, 其区别在于其中还包含了创建 JMX 对象名称的代码。有关 JMX 的内容将在第 20 章介绍。

变量 `errorReportValveClass` 的值定义在 `StandardHost` 类中:

```

private String errorReportValveClass =
    "org.apache.catalina.valves.ErrorReportValve";

```

每当引入一个 HTTP 请求, 都会调用 `Host` 实例的 `invoke` 方法。由于 `StandardHost` 类并没有提供 `invoke()` 方法的实现, 因此, 它会调用其父类 `ContainerBase` 类的 `invoke()` 方法。而 `ContainerBase` 的 `invoke()` 方法调用 `StandardHost` 实例的基础阀 `StandardHostValve` 实例的 `invoke()` 方法。`StandardHostValve` 类的 `invoke()` 方法将在 13.4 节讨论。此外, `StandardHostValve` 类的 `invoke()` 方法会调用 `StandardHost` 类的 `map()` 方法来获取相应的 `Context` 实例来处理 HTTP 请求。代码清单 13-3 给出了 `StandardHost` 类中 `map()` 方法的实现。

代码清单 13-3 `StandardHost` 类中 `map()` 方法的实现

```

public Context map(String uri) {
    if (debug > 0)
        log("Mapping request URI " + uri + "");
    if (uri == null)
        return (null);

    // Match on the longest possible context path prefix
    if (debug > 1)
        log(" Trying the longest context path prefix");
    Context context = null;
    String mapuri = uri;
    while (true) {
        context = (Context) findChild(mapuri);
        if (context != null)
            break;
        int slash = mapuri.lastIndexOf('/');
        if (slash < 0)
            break;
        mapuri = mapuri.substring(0, slash);
    }

    // If no Context matches, select the default Context
    if (context == null) {
        if (debug > 1)
            log(" Trying the default context");
    }
}

```

```

    context = (Context) findChild("");
}

// Complain if no Context has been selected
if (context == null) {
    log(sm.getString("standardHost.mappingError", uri));
    return (null);
}

// Return the mapped Context (if any)
if (debug > 0)
    log("Mapped to context '" + context.getPath() + "'");
return (context);
}

```

注意，在 Tomcat 4 中，ContainerBase 类也定义了一个 map() 方法，方法签名如下：

```
public Container map(Request request, boolean update);
```

在 Tomcat 4 中，StandardHostValve 类的 invoke() 方法会调用 ContainerBase 类的 map() 方法，而 map() 方法又会调用 StandardHost 类的 map() 方法。在 Tomcat 5 中，映射器组件已经移除，Context 实例是通过 request 对象获取的。

### 13.3 StandardHostMapper 类

在 Tomcat 4 中，ContainerBase 类（也就是 StandardHost 的父类）会调用其 addDefaultMapper() 方法创建一个默认映射器。默认映射器的类型由 mapperClass 属性的值决定。下面是 ContainerBase 类的 addDefaultMapper() 方法的实现：

```

protected void addDefaultMapper(String mapperClass) {
    // Do we need a default Mapper?
    if (mapperClass == null)
        return;
    if (mappers.size() >= 1)
        return;

    // Instantiate and add a default Mapper
    try {
        Class clazz = Class.forName(mapperClass);
        Mapper mapper = (Mapper) clazz.newInstance();
        mapper.setProtocol("http");
        addMapper(mapper);
    }
    catch (Exception e) {
        log(sm.getString("containerBase.addDefaultMapper", mapperClass),
            e);
    }
}

```

变量 mapperClass 的值定义在 StandardHost 类中：

```

private String mapperClass =
    "org.apache.catalina.core.StandardHostMapper";

```

此外，StandardHost 类的 start() 方法会在方法末尾调用父类的 start() 方法，确保默认映射器的创建完成。



**注意** 在 Tomcat 4 中, StandardContext 类创建默认映射器的方法略有不同。它的 start() 方法并不会调用父类的 start() 方法。相反, StandardContext 类的 start() 方法会调用 addDefaultMapper() 方法, 并传入 mapperClass 变量来创建默认映射器。

当然, StandardHostMapper 类中最重要的方法是 map() 方法。下面 map() 方法的实现:

```
public Container map(Request request, boolean update) {
    // Has this request already been mapped?
    if (update && (request.getContext() != null))
        return (request.getContext());

    // Perform mapping on our request URI
    String uri = ((HttpRequest) request).getDecodedRequestURI();
    Context context = host.map(uri);

    // Update the request (if requested) and return the selected Context
    if (update) {
        request.setContext(context);
        if (context != null)
            ((HttpRequest) request).setContextPath(context.getPath());
        else
            ((HttpRequest) request).setContextPath(null);
    }
    return (context);
}
```

注意, 这里 map() 方法仅仅是简单地调用了 Host 实例的 map() 方法。

## 13.4 StandardHostValve 类

org.apache.catalina.core.StandardHostValve 类是 StandardHost 实例的基础阀。当有引入的 HTTP 请求时, 会调用 StandardHostValve 类的 invoke() 方法对其进行处理。代码清单 13-4 给出了 invoke() 方法的实现。

代码清单 13-4 StandardHostValve 类的 invoke() 方法的实现

```
public void invoke(Request request, Response response,
    ValveContext valveContext)
    throws IOException, ServletException {

    // Validate the request and response object types
    if (!(request.getRequest() instanceof HttpServletRequest) ||
        !(response.getResponse() instanceof HttpServletResponse)) {
        return; // NOTE - Not much else we can do generically
    }

    // Select the Context to be used for this Request
    StandardHost host = (StandardHost) getContainer();
    Context context = (Context) host.map(request, true);
    if (context == null) {
        ((HttpServletResponse) response.getResponse()).sendError(
            HttpServletResponse.SC_INTERNAL_SERVER_ERROR,
            sm.getString("standardHost.noContext"));
        return;
    }

    // Bind the context CL to the current thread
```

```

Thread.currentThread().setContextClassLoader
(context.getLoader().getClassLoader());

// Update the session last access time for our session (if any)
HttpServletRequest hreq = (HttpServletRequest) request.getRequest();
String sessionId = hreq.getRequestedSessionId();
if (sessionId != null) {
    Manager manager = context.getManager();
    if (manager != null) {
        Session session = manager.findSession(sessionId);
        if ((session != null) && session.isValid())
            session.access();
    }
}

// Ask this Context to process this request
context.invoke(request, response);
}

```

在 Tomcat 4 中, `invoke()` 方法会调用 `StandardHost` 实例的 `map()` 方法来获取一个相应的 Context 实例:

```

// Select the Context to be used for this Request
StandardHost host = (StandardHost) getContainer();
Context context = (Context) host.map(request, true);

```

**注意** 在获取 Context 实例的代码中有一个往复的过程。上面的 `map()` 方法需要两个参数, 该方法定义在 `ContainerBase` 类中。`ContainerBase` 类中的 `map()` 方法会找到其子容器的映射器 (在本例中是 `StandardHost` 类的实例), 并调用其 `map()` 方法。

然后, `invoke()` 方法会获取与该 `request` 对象相关联的 `session` 对象, 并调用其 `access()` 方法。`access()` 方法会修改 `session` 对象的最后访问时间。下面是 `org.apache.catalina.session.StandardSession` 类中 `access()` 方法的实现:

```

public void access() {
    this.isNew = false;
    this.lastAccessedTime = this.thisAccessedTime;
    this.thisAccessedTime = System.currentTimeMillis();
}

```

最后, `invoke()` 方法调用 Context 实例的 `invoke()` 来处理 HTTP 请求。

## 13.5 为什么必须要有一个 Host 容器

在 Tomcat 4 和 5 的实际部署中, 若一个 Context 实例使用 `ContextConfig` 对象进行设置, 就必须使用一个 Host 对象。原因如下:

使用 `ContextConfig` 对象需要知道应用程序 `web.xml` 文件的位置。在其 `applicationConfig()` 方法中它会试图打开 `web.xml` 文件。下面是 `applicationConfig()` 方法的片段:

```

synchronized (webDigerster) {
    try {
        URL url =
            servletContext.getResource(Constants.ApplicationWebXml);
        InputSource is = new InputSource(url.toExternalForm());
        is.setByteStream(stream);
        ...
        webDigerster.parse(is);
        ...
    }
}

```

其中, Constants.ApplicationWebXml 的值为 “/WEB-INF/web.xml”, web.xml 文件的相对路径, servletContext 是一个 org.apache.catalina.core.ApplicationContext 类型 (实现了 javax.servlet.ServletContext 接口) 的对象。

下面是 ApplicationContext 类的 getResource() 方法的部分实现代码:

```
public URL getResource(String path)
    throws MalformedURLException {

    DirContext resources = context.getResources();
    if (resources != null) {
        String fullPath = context.getName() + path;
        // this is the problem. Host must not be null
        String hostName = context.getParent().getName();
```

其中, 最后一行清楚地表明了, 如果要使用 ContextConfig 实例来进行配置的话, Context 实例必须有一个 Host 实例作为其父容器。在第 15 章中, 你将学习到 web.xml 文件是如何解析的。简单来说, 除非你自己实现一个 ContextConfig 类, 否则, 你必须使用一个 Host 容器。

## 13.6 应用程序 1

本章的第 1 个应用程序重在说明 Host 容器作为顶层容器的使用方法。该应用程序包含两个类, 分别是 ex13.pyrmont.core.SimpleContextConfig 类和 ex13.pyrmont.startup.Bootstrap1 类。其中 SimpleContextConfig 类与第 11 章中的相同。代码清单 13-5 给出了 Bootstrap1 类的定义。

代码清单 13-5 Bootstrap1 类

```
package ex13.pyrmont.startup;

import ex13.pyrmont.core.SimpleContextConfig;
import org.apache.catalina.Connector;
import org.apache.catalina.Context;
import org.apache.catalina.Host;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleListener;
import org.apache.catalina.Loader;
import org.apache.catalina.Wrapper;
import org.apache.catalina.connector.http.HttpConnector;
import org.apache.catalina.core.StandardContext;
import org.apache.catalina.core.StandardHost;
import org.apache.catalina.core.StandardWrapper;
import org.apache.catalina.loader.WebappLoader;

public final class Bootstrap1 {
    public static void main(String[] args) {
        System.setProperty("catalina.base",
            System.getProperty("user.dir"));
        Connector connector = new HttpConnector();
        Wrapper wrapper1 = new StandardWrapper();
        wrapper1.setName("Primitive");
        wrapper1.setServletClass("PrimitiveServlet");
        Wrapper wrapper2 = new StandardWrapper();
        wrapper2.setName("Modern");
        wrapper2.setServletClass("ModernServlet");
        Context context = new StandardContext();
        // StandardContext's start method adds a default mapper
        context.setPath("/app1");
```



```

context.setDocBase("app1");
context.addChild(wrapper1);
context.addChild(wrapper2);
LifecycleListener listener = new SimpleContextConfig();
((Lifecycle) context).addLifecycleListener(listener);
Host host = new StandardHost();
host.addChild(context);
host.setName("localhost");
host.setAppBase("webapps");
Loader loader = new WebappLoader();
context.setLoader(loader);
// context.addServletMapping(pattern, name);
context.addServletMapping("/Primitive", "Primitive");
context.addServletMapping("/Modern", "Modern");
connector.setContainer(host);
try {
    connector.initialize();
    ((Lifecycle) connector).start();
    ((Lifecycle) host).start();
    // make the application wait until we press a key.
    System.in.read();
    ((Lifecycle) host).stop();
}
catch (Exception e) {
    e.printStackTrace();
}
}
}

```

## 运行应用程序

要在 Windows 平台下运行应用程序，需要在工作目录中执行以下命令：

```

java -classpath ./lib/servlet.jar;./lib/commons-
collections.jar;./lib/commons-digester.jar;./
ex13.pyrmont.startup.Bootstrap1

```

而在 Linux 平台下，需要使用冒号来分隔不同库：

```

java -classpath ./lib/servlet.jar:./lib/commons-
collections.jar:./lib/commons-digester.jar:./
ex13.pyrmont.startup.Bootstrap1

```

要调用 PrimitiveServlet，需要在浏览器中输入以下 URL：

```
http://localhost:8080/app1/Primitive
```

要调用 ModernServlet，需要在浏览器中输入以下 URL：

```
http://localhost:8080/app1/Modern
```

## 13.7 Engine 接口

Engine 容器是 org.apache.catalina.Engine 接口的实例。Engine 容器也就是 Tomcat 的 servlet 引擎。当部署 Tomcat 时要支持多个虚拟机的话，就需要使用 Engine 容器。事实上，一般情况下，部署的 Tomcat 都会使用一个 Engine 容器。

代码清单 13-6 给出了 Engine 接口的定义。

代码清单 13-6 Engine 接口的定义

```

package org.apache.catalina;

public interface Engine extends Container {
    /**
     * Return the default hostname for this Engine.
     */
    public String getDefaultHost();
    /**
     * Set the default hostname for this Engine.
     *
     * @param defaultHost The new default host
     */
    public void setDefaultHost(String defaultHost);
    /**
     * Retrieve the JvmRouteId for this engine.
     */
    public String getJvmRoute();
    /**
     * Set the JvmRouteId for this engine.
     *
     * @param jvmRouteId the (new) JVM Route ID. Each Engine within a
     * cluster must have a unique JVM Route ID.
     */
    public void setJvmRoute(String jvmRouteId);
    /**
     * Return the <code>Service</code> with which we are associated (if
     * any).
     */
    public Service getService();
    /**
     * Set the <code>Service</code> with which we are associated (if
     * any).
     *
     * @param service The service that owns this Engine
     */
    public void setService(Service service);
    /**
     * Set the DefaultContext
     * for new web applications.
     *
     * @param defaultContext The new DefaultContext
     */
    public void addDefaultContext(DefaultContext defaultContext);
    /**
     * Retrieve the DefaultContext for new web applications.
     */
    public DefaultContext getDefaultContext();
    /**
     * Import the DefaultContext config into a web application context.
     *
     * @param context web application context to import default context
     */
    public void importDefaultContext(Context context);
}

```

在 Engine 容器中，可以设置一个默认的 Host 容器或一个默认的 Context 容器。注意，Engine 容器可以与一个服务实例相关联。服务将在第 14 章讨论。

## 13.8 StandardEngine 类

org.apache.catalina.core.StandardEngine 类是 Engine 接口的标准实现。相比于 StandardContext 类和 StandardHost 类, StandardEngine 类相对小一些。在实例化的时候, StandardEngine 类会添加一个基础阀, 如其构造函数所示:

```
public StandardEngine() {
    super();
    pipeline.setBasic(new StandardEngineValve());
}
```

作为一个顶层容器, Engine 容器可以有子容器, 而它的子容器只能是 Host 容器, 所以, 若是给它设置一个非 Host 类型的容器, 就会抛出异常。下面是 StandardEngine 类的 addChild() 方法的实现代码:

```
public void addChild(Container child) {
    if (!(child instanceof Host))
        throw new IllegalArgumentException(
            (sm.getString("standardEngine.notHost")));
    super.addChild(child);
}
```

此外, Engine 容器也不能再有父容器了。如果调用 StandardEngine 类的 setContainer() 方法, 为其添加一个父容器时, 就会抛出异常:

```
public void setParent(Container container) {
    throw new IllegalArgumentException(
        (sm.getString("standardEngine.notParent")));
}
```

## 13.9 StandardEngineValve 类

org.apache.catalina.core.StandardEngineValve 类是 StandardEngine 容器的基础阀。代码清单 13-7 给出了 StandardEngineValve 类的 invoke() 方法的实现代码。

代码清单 13-7 StandardEngineValve 类的 invoke() 方法的实现

```
public void invoke(Request request, Response response,
    ValveContext valveContext)
    throws IOException, ServletException {
    // Validate the request and response object types
    if (!(request.getRequest() instanceof HttpServletRequest) ||
        !(response.getResponse() instanceof HttpServletResponse)) {
        return; // NOTE - Not much else we can do generically
    }
    // Validate that any HTTP/1.1 request included a host header
    HttpServletRequest hrequest = (HttpServletRequest) request;
    if ("HTTP/1.1".equals(hrequest.getProtocol()) &&
        (hrequest.getServerName() == null)) {
        ((HttpServletResponse) response.getResponse()).sendError(
            (HttpServletResponse.SC_BAD_REQUEST,
                sm.getString("standardEngine.noHostHeader",
                    request.getRequest().getServerName())));
        return;
    }
    // Select the Host to be used for this Request
    StandardEngine engine = (StandardEngine) getContainer();
```



```

Host host = (Host) engine.map(request, true);
if (host == null) {
    ((HttpServletResponse) response.getResponse()).sendError
    (HttpServletResponse.SC_BAD_REQUEST,
    sm.getString("standardEngine.noHost",
    request.getRequest().getServerName()));
    return;
}
// Ask this Host to process this request
host.invoke(request, response);
}

```

在验证了 request 和 response 对象的类型后, invoke() 方法得到 Host 实例, 用于处理该请求。invoke() 方法会通过调用 Engine 实例的 map() 方法获取 Host 对象。得到 Host 对象以后, 调用其 invoke() 方法处理请求。

## 13.10 应用程序 2

本章的第 2 个应用程序重在说明如何作为顶层容器使用 Engine 容器。该应用程序包含两个类, 分别是 ex13.pyrmont.core.SimpleContextConfig 类和 ex13.pyrmont.startup.Bootstrap2 类。代码清单 13-8 给出了 Bootstrap2 类的实现代码。

代码清单 13-8 Bootstrap2 类的定义

```

package ex13.pyrmont.startup;
//Use engine
import ex13.pyrmont.core.SimpleContextConfig;
import org.apache.catalina.Connector;
import org.apache.catalina.Context;
import org.apache.catalina.Engine;
import org.apache.catalina.Host;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleListener;
import org.apache.catalina.Loader;
import org.apache.catalina.Wrapper;
import org.apache.catalina.connector.http.HttpConnector;
import org.apache.catalina.core.StandardContext;
import org.apache.catalina.core.StandardEngine;
import org.apache.catalina.core.StandardHost;
import org.apache.catalina.core.StandardWrapper;
import org.apache.catalina.loader.WebappLoader;

public final class Bootstrap2 {
    public static void main(String[] args) {
        System.setProperty("catalina.base",
            System.getProperty("user.dir"));
        Connector connector = new HttpConnector();
        Wrapper wrapper1 = new StandardWrapper();
        wrapper1.setName("Primitive");
        wrapper1.setServletClass("PrimitiveServlet");
        Wrapper wrapper2 = new StandardWrapper();
        wrapper2.setName("Modern");
        wrapper2.setServletClass("ModernServlet");
        Context context = new StandardContext();
        // StandardContext's start method adds a default mapper
        context.setPath("/app1");
    }
}

```

```

context.setDocBase("app1");
context.addChild(wrapper1);
context.addChild(wrapper2);
LifecycleListener listener = new SimpleContextConfig();
((Lifecycle) context).addLifecycleListener(listener);
Host host = new StandardHost();
host.addChild(context);
host.setName("localhost");
host.setAppBase("webapps");
Loader loader = new WebappLoader();
context.setLoader(loader);
// context.addServletMapping(pattern, name);
context.addServletMapping("/Primitive", "Primitive");
context.addServletMapping("/Modern", "Modern");
Engine engine = new StandardEngine();
engine.addChild(host);
engine.setDefaultHost("localhost");
connector.setContainer(engine);
try {
    connector.initialize();
    ((Lifecycle) connector).start();
    ((Lifecycle) engine).start();
    // make the application wait until we press a key.
    System.in.read();
    ((Lifecycle) engine).stop();
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

## 运行应用程序

要在 Windows 平台运行应用程序，需要在工作目录中执行以下命令：

```

java -classpath ./lib/servlet.jar;./lib/commons-
collections.jar;./lib/commons-digester.jar;./
ex13.pyrmont.startup.Bootstrap2

```

而在 Linux 平台下，需要使用冒号来分隔不同库：

```

java -classpath ./lib/servlet.jar:./lib/commons-
collections.jar:./lib/commons-digester.jar:./
ex13.pyrmont.startup.Bootstrap2

```

要调用 PrimitiveServlet，需要在浏览器中输入以下 URL：

```
http://localhost:8080/app1/Primitive
```

要调用 ModernServlet，需要在浏览器中输入以下 URL：

```
http://localhost:8080/app1/Modern
```

## 13.11 小结

在本章中，你已经学习了两种类型的容器，分别是 Host 和 Engine，并介绍了与其相关的实现类。本章中的两个应用程序分别展示了 Host 容器和 Engine 容器作为顶层容器的使用方法。

## 第 14 章

# 服务器组件和服务组件

在前面的章节中，你已经学会了如何通过实例化一个连接器和容器来获得一个 servlet 容器，并将连接器和容器相关联。但在前面的章节中只有一个连接器可以使用，该连接器服务 8080 端口上的 HTTP 请求。无法添加另一个连接器来服务诸如 HTTPS 之类的其他请求。

此外，在前面章节的应用程序中有些缺憾，即缺少一种启动 / 关闭 servlet 容器的机制。在本章中，我们会学习两种提供这些特性的组件，分别是服务器组件和服务组件。

### 14.1 服务器组件

org.apache.catalina.Server 接口的实例表示 Catalina 的整个 servlet 引擎，囊括了所有的组件。服务器组件是非常有用的，因为它使用了一种优雅的方法来启动 / 关闭整个系统，不需要再对连接器和容器分别启动 / 关闭。

下面说明一下启动 / 关闭机制的具体工作原理。当启动服务器组件时，它会启动其中所有的组件，然后它就无限期地等待关闭命令。如果想要关闭系统，可以向指定端口发送一条关闭命令。服务器组件接收到关闭命令后，就会关闭其中所有的组件。

服务器组件使用了另一个组件（即服务组件）来包含其他组件，如一个容器组件和一个 / 多个连接器组件。服务组件将会在 14.3 节介绍。

代码清单 14-1 给出了 Server 接口的定义。

代码清单 14-1 Server 接口的定义

```
package org.apache.catalina;
import org.apache.catalina.deploy.NamingResources;

public interface Server {
    /**
     * Return descriptive information about this Server implementation
     * and the corresponding version number, in the format
     * <code><description><version></code>.
     */
    public String getInfo();
    /**
     * Return the global naming resources.
     */
    public NamingResources getGlobalNamingResources();
    /**
     * Set the global naming resources.
     *
     * @param namingResources The new global naming resources
     */
    public void setGlobalNamingResources
```



```

(NamingResources globalNamingResources);
/**
 * Return the port number we listen to for shutdown commands.
 */
public int getPort();
/**
 * Set the port number we listen to for shutdown commands.
 *
 * @param port The new port number
 */
public void setPort(int port);
/**
 * Return the shutdown command string we are waiting for.
 */
public String getShutdown();
/**
 * Set the shutdown command we are waiting for.
 *
 * @param shutdown The new shutdown command
 */
public void setShutdown(String shutdown);
/**
 * Add a new Service to the set of defined Services.
 *
 * @param service The Service to be added
 */
public void addService(Service service);
/**
 * Wait until a proper shutdown command is received, then return.
 */
public void await();
/**
 * Return the specified Service (if it exists); otherwise return
 * <code>null</code>.
 *
 * @param name Name of the Service to be returned
 */
public Service findService(String name);
/**
 * Return the set of Services defined within this Server.
 */
public Service[] findServices();
/**
 * Remove the specified Service from the set associated from this
 * Server.
 *
 * @param service The Service to be removed
 */
public void removeService(Service service);
/**
 * Invoke a pre-startup initialization. This is used to allow
 * connectors to bind to restricted ports under Unix operating
 * environments.
 *
 * @exception LifecycleException If this server was already
 * initialized.
 */
public void initialize() throws LifecycleException;
}

```

shutdown 属性保存了必须发送给 Server 实例用来关闭整个系统的关闭命令。port 属性定义了服务器组件会从哪里获取关闭命令。可以调用其 addService() 方法为服务器组件添加服务组件, 或通过 removeService() 方法删除某个服务组件。findServices() 方法将会返回添加到该服务器组件中的所有服务组件。initialize() 方法包含在系统启动前要执行的一些代码。

## 14.2 StandardServer 类

org.apache.catalina.core.StandardServer 类是 Server 接口的标准实现。介绍这个类是因为我们对其中的关闭机制比较感兴趣, 而这也是这个类中最重要的特性。该类中的许多方法都与新 server.xml 文件中的服务器配置的存储相关, 但这些并不是本章的重点。如果你感兴趣的话, 不妨自己学习一下, 它们并不难于理解。

一个服务器组件可以有 0 个或多个服务组件。StandardServer 类提供了 addService() 方法、removeService() 方法和 findServices() 方法的实现。

StandardServer 类有 4 个与生命周期相关的方法, 分别是 initialize() 方法、start() 方法、stop() 方法和 await() 方法。就像其他组件一样, 可以初始化并启动服务器组件, 也可以调用 await() 方法和 stop() 方法。调用 await() 方法后会一直阻塞住, 直到它从 8085 端口 (也可以配置其他端口) 上接收到关闭命令。当 await() 方法返回时, 会运行 stop() 方法来关闭其下的所有子组件。在本章的应用程序中, 你会学习到这种关闭机制是如何实现的。

initialize() 方法、start() 方法、stop() 方法和 await() 方法将在下面几节分别讨论。

### 14.2.1 initialize() 方法

服务器组件的 initialize() 方法用于初始化添加到其中的服务器组件。代码清单 14-2 给出了 Tomcat 4 中 StandardServer 类中 initialize() 方法的实现。

代码清单 14-2 StandardServer 类中 initialize() 方法的实现

```
public void initialize() throws LifecycleException {
    if (initialized)
        throw new LifecycleException (
            sm.getString("standardServer.initialize.initialized"));
    initialized = true;

    // Initialize our defined Services
    for (int i = 0; i < services.length; i++) {
        services[i].initialize();
    }
}
```

注意, initialize() 方法使用一个名为 initialized 的布尔变量来防止服务器组件初始化两次。在 Tomcat 5 中, initialize() 方法类似, 但它也包含一些与 JMX 相关的代码 (JMX 将在第 20 章中介绍)。stop() 方法并没有重置布尔变量 initialized 的值, 所以如果服务器组件关闭后再启动, 是不会有再次进行初始化的。

## 14.2.2 start() 方法

start() 方法用于启动服务器组件。在 StandardServer 类的 start() 方法的实现中，它会启动其所有的服务组件，逐个启动所有的组件，如连接器组件和 servlet 容器。代码清单 14-3 给出了 start() 方法的实现。

代码清单 14-3 start() 方法的实现

```
public void start() throws LifecycleException {
    // Validate and update our current component state
    if (started)
        throw new LifecycleException
            (sm.getString("standardServer.start.started"));
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);
    lifecycle.fireLifecycleEvent(START_EVENT, null);
    started = true;
    // Start our defined Services
    synchronized (services) {
        for (int i = 0; i < services.length; i++) {
            if (services[i] instanceof Lifecycle)
                ((Lifecycle) services[i]).start();
        }
    }
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(AFTER_START_EVENT, null);
}
```

start() 方法使用布尔变量 start 来防止服务器组件重复启动。stop() 方法会重置这个变量。

## 14.2.3 stop() 方法

stop() 方法用于关闭服务器组件。代码清单 14-4 给出了 stop() 方法的实现。

代码清单 14-4 stop() 方法的实现

```
public void stop() throws LifecycleException {
    // Validate and update our current component state
    if (!started)
        throw new LifecycleException
            (sm.getString("standardServer.stop.notStarted"));
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(BEFORE_STOP_EVENT, null);
    lifecycle.fireLifecycleEvent(STOP_EVENT, null);
    started = false;
    // Stop our defined Services
    for (int i = 0; i < services.length; i++) {
        if (services[i] instanceof Lifecycle)
            ((Lifecycle) services[i]).stop();
    }
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(AFTER_STOP_EVENT, null);
}
```

调用 stop() 方法会关闭所有的服务组件，并重置布尔变量 started，这样，才能再次启动服务器组件。



### 14.2.4 await() 方法

await() 方法负责等待关闭整个 Tomcat 部署的命令。代码清单 14-5 给出了 await() 方法的实现。

代码清单 14-5 await() 方法的实现

```

/**
 * Wait until a proper shutdown command is received, then return.
 */
public void await() {
    // Set up a server socket to wait on
    ServerSocket serverSocket = null;
    try {
        serverSocket = new ServerSocket(port, 1,
            InetAddress.getByAddress("127.0.0.1"));
    } catch (IOException e) {
        System.err.println("StandardServer.await: create[" +
            port + "]: " + e);
        e.printStackTrace();
        System.exit(1);
    }
    // Loop waiting for a connection and a valid command
    while (true) {
        // Wait for the next connection
        Socket socket = null;
        InputStream stream = null;
        try {
            socket = serverSocket.accept();
            socket.setSoTimeout(10 * 1000); // Ten seconds
            stream = socket.getInputStream();
        } catch (AccessControlException ace) {
            System.err.println("StandardServer.accept security exception: "
                + ace.getMessage());
            continue;
        } catch (IOException e) {
            System.err.println("StandardServer.await: accept: " + e);
            e.printStackTrace();
            System.exit(1);
        }

        // Read a set of characters from the socket
        StringBuffer command = new StringBuffer();
        int expected = 1024; // Cut off to avoid DoS attack
        while (expected < shutdown.length()) {
            if (random == null)
                random = new Random(System.currentTimeMillis());
            expected += (random.nextInt() % 1024);
        }
        while (expected > 0) {
            int ch = -1;
            try {
                ch = stream.read();
            } catch (IOException e) {
                System.err.println("StandardServer.await: read: " + e);
                e.printStackTrace();
                ch = -1;
            }
        }
    }
}

```

```

    if (ch < 32) // Control character or EOF terminates loop
        break;
    command.append((char) ch);
    expected--;
}

// Close the socket now that we are done with it
try {
    socket.close();
}
catch (IOException e) {
    ;
}

// Match against our command string
boolean match = command.toString().equals(shutdown);
if (match) {
    break;
}
else
    System.err.println("StandardServer.await: Invalid command '" +
        command.toString() + "' received");
}

// Close the server socket and return
try {
    serverSocket.close();
}
catch (IOException e) {
    ;
}
}

```

await() 方法创建一个 ServerSocket 对象，监听 8085 端口，并在 while 循环中调用它的 accept() 方法。当在指定端口上接收到消息时，才会从 accept() 方法中返回。然后将接收到的消息与关闭命令的字符串相比较，相同的话就跳出 while 循环，关闭 SocketServer，否则会再次循环，继续等待消息。

### 14.3 Service 接口

服务组件是 org.apache.catalina.Service 接口的实例。一个服务组件可以有一个 servlet 容器和多个连接器实例。可以自由地把连接器实例添加到服务组件中，所有的连接器都会与这个 servlet 容器相关联。代码清单 14-6 给出了 Service 接口的定义。

代码清单 14-6 Service 接口的定义

```

package org.apache.catalina;

public interface Service {

    /**
     * Return the <code>Container</code> that handles requests for all
     * <code>Connectors</code> associated with this Service.
     */
    public Container getContainer();
}

```

```

/**
 * Set the <code>Container</code> that handles requests for all
 * <code>Connectors</code> associated with this Service.
 *
 * @param container The new Container
 */
public void setContainer(Container container);

/**
 * Return descriptive information about this Service implementation
 * and the corresponding version number, in the format
 * <code>&lt;description&gt;&lt;version&gt;</code>.
 */
public String getInfo();

/**
 * Return the name of this Service.
 */
public String getName();

/**
 * Set the name of this Service.
 *
 * @param name The new service name
 */
public void setName(String name);

/**
 * Return the <code>Server</code> with which we are associated
 * (if any).
 */
public Server getServer();

/**
 * Set the <code>Server</code> with which we are associated (if any).
 *
 * @param server The server that owns this Service
 */
public void setServer(Server server);

/**
 * Add a new Connector to the set of defined Connectors,
 * and associate it with this Service's Container.
 *
 * @param connector The Connector to be added
 */
public void addConnector(Connector connector);

/**
 * Find and return the set of Connectors associated with
 * this Service.
 */
public Connector[] findConnectors();

/**
 * Remove the specified Connector from the set associated from this
 * Service. The removed Connector will also be disassociated
 * from our Container.
 *
 * @param connector The Connector to be removed
 */
public void removeConnector(Connector connector);

```



```

/**
 * Invoke a pre-startup initialization. This is used to
 * allow connectors to bind to restricted ports under
 * Unix operating environments.
 *
 * @exception LifecycleException If this server was
 * already initialized.
 */
public void initialize() throws LifecycleException;
}

```

## 14.4 StandardService 类

org.apache.catalina.core.StandardService 类是 Service 接口的标准实现。StandardService 类的 initialize() 方法用于初始化添加到其中的所有连接器。此外，StandardService 类还实现 Service 以及 org.apache.catalina.Lifecycle 接口，因此，它的 start() 方法也可以启动连接器和所有 servlet 容器。

### 14.4.1 connector 和 container

StandardService 实例中有两种组件，分别是连接器和 servlet 容器。其中 servlet 容器只有一个，而连接器则可以有多多个。多个连接器使 Tomcat 可以为多种不同的请求协议提供服务。例如，一个连接器处理 HTTP 请求，而另一个可以处理 HTTPS 请求。

StandardService 类使用变量 container 来指向一个 Container 接口的实例，使用数组 connectors 来保存所有连接器的引用：

```

private Container container = null;
private Connector connectors[] = new Connector[0];

```

需要调用 setContainer() 方法将 servlet 容器与服务组件相关联。代码清单 14-7 给出了 setContainer() 方法的实现。

代码清单 14-7 setContainer() 方法的实现

```

public void setContainer(Container container) {
    Container oldContainer = this.container;
    if ((oldContainer != null) && (oldContainer instanceof Engine))
        ((Engine) oldContainer).setService(null);
    this.container = container;
    if ((this.container != null) && (this.container instanceof Engine))
        ((Engine) this.container).setService(this);
    if (started && (this.container != null) &&
        (this.container instanceof Lifecycle)) {
        try {
            ((Lifecycle) this.container).start();
        }
        catch (LifecycleException e) {
            ;
        }
    }
    synchronized (connectors) {
        for (int i = 0; i < connectors.length; i++)
            connectors[i].setContainer(this.container);
    }
}

```

```

if (started && (oldContainer != null) &&
    (oldContainer instanceof Lifecycle)) {
    try {
        ((Lifecycle) oldContainer).stop();
    }
    catch (LifecycleException e) {
        ;
    }
}

// Report this property change to interested listeners
support.firePropertyChange("container", oldContainer,
    this.container);
}

```

与服务组件相关联的 servlet 容器的实例将被传给每个连接器对象的 setContainer() 方法，这样在服务组件中就可以形成每个连接器和 servlet 容器之间的关联关系。

可以调用 addConnector() 方法将连接器添加到服务组件中，调用 removeConnector() 方法将某个连接器移除。代码清单 14-8 给出了 addConnector() 方法的实现，代码清单 14-9 给出了 removeConnector() 方法的实现。

代码清单 14-8 addConnector() 方法的实现

```

public void addConnector(Connector connector) {
    synchronized (connectors) {
        connector.setContainer(this.container);
        connector.setService(this);
        Connector results[] = new Connector[connectors.length + 1];
        System.arraycopy(connectors, 0, results, 0, connectors.length);
        results[connectors.length] = connector;
        connectors = results;

        if (initialized) {
            try {
                connector.initialize();
            }
            catch (LifecycleException e) {
                e.printStackTrace(System.err);
            }
        }

        if (started && (connector instanceof Lifecycle)) {
            try {
                ((Lifecycle) connector).start();
            }
            catch (LifecycleException e) {
                ;
            }
        }

        // Report this property change to interested listeners
        support.firePropertyChange("connector", null, connector);
    }
}

```

代码清单 14-9 removeConnector() 方法的实现

```

public void removeConnector(Connector connector) {
    synchronized (connectors) {
        int j = -1;
        for (int i = 0; i < connectors.length; i++) {
            if (connector == connectors[i]) {
                j = i;
                break;
            }
        }
        if (j < 0)
            return;
        if (started && (connectors[j] instanceof Lifecycle)) {
            try {
                ((Lifecycle) connectors[j]).stop();
            }
            catch (LifecycleException e) {
            }
        }
        connectors[j].setContainer(null);
        connector.setService(null);
        int k = 0;
        Connector results[] = new Connector[connectors.length - 1];
        for (int i = 0; i < connectors.length; i++) {
            if (i != j)
                results[k++] = connectors[i];
        }
        connectors = results;
        // Report this property change to interested listeners
        support.firePropertyChange("connector", connector, null);
    }
}

```

addConnector() 方法会初始化并启动添加到其中的连接器。

#### 14.4.2 与生命周期有关的方法

与生命周期有关的方法包括从 Lifecycle 接口中实现的 start() 和 stop() 方法，再加上 initialize() 方法。其中 initialize() 方法会调用该服务组件中所有连接器的 initialize() 方法。代码清单 14-10 给出了 initialize() 方法的实现。

代码清单 14-10 initialize() 方法的实现

```

public void initialize() throws LifecycleException {
    if (initialized)
        throw new LifecycleException (
            sm.getString("standardService.initialize.initialized"));
    initialized = true;
    // Initialize our defined Connectors
    synchronized (connectors) {
        for (int i = 0; i < connectors.length; i++) {
            connectors[i].initialize();
        }
    }
}

```



start() 方法负责启动被添加到该服务组件中的连接器和 servlet 容器。代码清单 14-11 给出了 start() 方法的实现。

代码清单 14-11 start() 方法的实现

```
public void start() throws LifecycleException {
    // Validate and update our current component state
    if (started) {
        throw new LifecycleException
            (sm.getString("standardService.start.started"));
    }

    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);

    System.out.println
        (sm.getString("standardService.start.name", this.name));
    lifecycle.fireLifecycleEvent(START_EVENT, null);
    started = true;

    // Start our defined Container first
    if (container != null) {
        synchronized (container) {
            if (container instanceof Lifecycle) {
                ((Lifecycle) container).start();
            }
        }

    }

    // Start our defined Connectors second
    synchronized (connectors) {
        for (int i = 0; i < connectors.length; i++) {
            if (connectors[i] instanceof Lifecycle)
                ((Lifecycle) connectors[i]).start();
        }
    }

    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(AFTER_START_EVENT, null);
}
```

stop() 方法用于关闭与该服务组件相关联的 servlet 容器和所有连接器。代码清单 14-12 给出了 stop() 方法的实现。

代码清单 14-12 stop() 方法的实现

```
public void stop() throws LifecycleException {
    // Validate and update our current component state
    if (!started) {
        throw new LifecycleException
            (sm.getString("standardService.stop.notStarted"));
    }

    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(BEFORE_STOP_EVENT, null);

    lifecycle.fireLifecycleEvent(STOP_EVENT, null);

    System.out.println
```

```

(sm.getString("standardService.stop.name", this.name));
started = false;

// Stop our defined Connectors first
synchronized (connectors) {
    for (int i = 0; i < connectors.length; i++) {
        if (connectors[i] instanceof Lifecycle)
            ((Lifecycle) connectors[i]).stop();
    }
}

// Stop our defined Container second
if (container != null) {
    synchronized (container) {
        if (container instanceof Lifecycle) {
            ((Lifecycle) container).stop();
        }
    }
}

// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(AFTER_STOP_EVENT, null);
}

```

## 14.5 应用程序

本章的应用程序重在说明如何使用服务器组件和服务组件，特别是在 `StandardServer` 类中如何利用启动和关闭机制。应用程序主要使用三个类，`SimpleContextConfig`（与第 13 章中应用程序的一个副本相同），`Bootstrap`（用于启动程序）和 `Stopper`（用于关闭程序）。

### 14.5.1 Bootstrap 类

代码清单 14-13 给出了 `Bootstrap` 类的定义。

代码清单 14-13 `Bootstrap` 类的定义

```

package ex14.pyrmont.startup;

import ex14.pyrmont.core.SimpleContextConfig;
import org.apache.catalina.Connector;
import org.apache.catalina.Context;
import org.apache.catalina.Engine;
import org.apache.catalina.Host;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleException;
import org.apache.catalina.LifecycleListener;
import org.apache.catalina.Loader;
import org.apache.catalina.Server;
import org.apache.catalina.Service;
import org.apache.catalina.Wrapper;
import org.apache.catalina.connector.http.HttpConnector;
import org.apache.catalina.core.StandardContext;
import org.apache.catalina.core.StandardEngine;
import org.apache.catalina.core.StandardHost;
import org.apache.catalina.core.StandardServer;
import org.apache.catalina.core.StandardService;

```

```

import org.apache.catalina.core.StandardWrapper;
import org.apache.catalina.loader.WebappLoader;

public final class Bootstrap {
    public static void main(String[] args) {
        System.setProperty("catalina.base",
            System.getProperty("user.dir"));
        Connector connector = new HttpConnector();
        Wrapper wrapper1 = new StandardWrapper();
        wrapper1.setName("Primitive");
        wrapper1.setServletClass("PrimitiveServlet");
        Wrapper wrapper2 = new StandardWrapper();
        wrapper2.setName("Modern");
        wrapper2.setServletClass("ModernServlet");
        Context context = new StandardContext();
        // StandardContext's start method adds a default mapper
        context.setPath("/app1");
        context.setDocBase("app1");
        context.addChild(wrapper1);
        context.addChild(wrapper2);
        LifecycleListener listener = new SimpleContextConfig();
        ((Lifecycle) context).addLifecycleListener(listener);
        Host host = new StandardHost();
        host.addChild(context);
        host.setName("localhost");
        host.setAppBase("webapps");
        Loader loader = new WebappLoader();
        context.setLoader(loader);
        // context.addServletMapping(pattern, name);
        context.addServletMapping("/Primitive", "Primitive");
        context.addServletMapping("/Modern", "Modern");
        Engine engine = new StandardEngine();
        engine.addChild(host);
        engine.setDefaultHost("localhost");
        Service service = new StandardService();
        service.setName("Stand-alone Service");
        Server server = new StandardServer();
        server.addService(service);
        service.addConnector(connector);
        // StandardService class's setContainer method calls
        // its connectors' setContainer method
        service.setContainer(engine);
        // Start the new server
        if (server instanceof Lifecycle) {
            try {
                server.initialize();
                ((Lifecycle) server).start();
                server.await();
                // the program waits until the await method returns,
                // i.e. until a shutdown command is received.
            }
            catch (LifecycleException e) {
                e.printStackTrace(System.out);
            }
        }

        // Shut down the server
        if (server instanceof Lifecycle) {
            try {
                ((Lifecycle) server).stop();
            }
            catch (LifecycleException e) {

```



```
e.printStackTrace(System.out);
```

```
}
```

```
}
```

Bootstrap 类的 `main()` 方法的开始部分与第 13 章中的类似。它会创建一个连接器、两个 Wrapper 实例、一个 Context 容器、一个 Host 容器和一个 Engine 容器。然后，将两个 Wrapper 实例添加到 Context 容器中，将 Context 容器添加到 Host 容器中，在将 Host 容器添加到 Engine 容器中。但是，它并没有将连接器与顶层 servlet 容器，也就是 Engine 容器相关联。相反，`main()` 方法创建一个 Service 对象，设置其名称，再创建一个 Server 对象，将 Service 实例添加到 Server 实例中：

```
Service service = new StandardService();
service.setName("Stand-alone Service");
Server server = new StandardServer();
server.addService(service);
```

然后，`main()` 方法将连接器和 Engine 容器添加到 Service 实例中：

```
service.addConnector(connector);
service.setContainer(engine);
```

这样，连接器就和 servlet 容器在 Service 实例中关联起来了。

然后，`main()` 方法调用 Server 实例的 `initialize()` 和 `start()` 方法，初始化连接器，并启动连接器和 servlet 容器：

```
if (server instanceof Lifecycle) {
    try {
        server.initialize();
        ((Lifecycle) server).start();
    }
```

接下来，`main()` 方法调用 Server 实例的 `await()` 方法，进入循环等待，监听 8085 端口等待关闭命令。注意，此时连接器已经处在运行状态，等待另一个端口 8080（默认端口）上的 HTTP 请求：

```
server.await();
```

除非接收到了正确的关闭命令，否则 `await()` 方法是不会返回的。当 `await()` 返回时，`main()` 方法调用 Server 实例的 `stop()` 方法，实际上关闭其所有组件。

下面再看一下用来关闭 Server 实例的 Stopper 类的定义。

## 14.5.2 Stopper 类

在前些章节的应用程序中，通过按某个键或强制中断的方式关闭 servlet 容器。Stopper 类提供了一种更优雅的方式来关闭 Catalina 服务器。此外，它也保证了所有的生命周期组件的 `stop()` 方法都能够调用。

代码清单 14-14 给出了 Stopper 类的定义。

代码清单 14-14 Stopper 类

```

package ex14.pyrmont.startup;

import java.io.OutputStream;
import java.io.IOException;
import java.net.Socket;

public class Stopper {
    public static void main(String[] args) {
        // the following code is taken from the Stop method of
        // the org.apache.catalina.startup.Catalina class
        int port = 8005;
        try {
            Socket socket = new Socket("127.0.0.1", port);
            OutputStream stream = socket.getOutputStream();
            String shutdown = "SHUTDOWN";
            for (int i = 0; i < shutdown.length(); i++)
                stream.write(shutdown.charAt(i));
            stream.flush();
            stream.close();
            socket.close();
            System.out.println("The server was successfully shut down.");
        }
        catch (IOException e) {
            System.out.println("Error. The server has not been started.");
        }
    }
}

```

Stopper 类的 main() 方法会创建一个 Socket 对象，然后将正确的关闭命令“SHUTDOWN”字符串发送给端口 8085。Catalina 服务器在接收到关闭命令后，就会执行相应的关闭操作。

### 14.5.3 运行应用程序

要在 Windows 平台下运行应用程序，需要在工作目录中执行以下命令：

```

java -classpath ./lib/servlet.jar;./lib/commons-
collections.jar;./lib/commons-digester.jar;./lib/naming-
factory.jar;./lib/naming-common.jar;./ ex14.pyrmont.startup.Bootstrap

```

而在 Linux 平台下，需要使用冒号来分隔不同库：

```

java -classpath ./lib/servlet.jar:./lib/commons-
collections.jar:./lib/commons-digester.jar:./lib/naming-
factory.jar:./lib/naming-common.jar:./ ex14.pyrmont.startup.Bootstrap

```

要调用 PrimitiveServlet，需要在浏览器中输入以下 URL：

http://localhost:8080/app/Primitive

要调用 ModernServlet，需要在浏览器中输入以下 URL：

http://localhost:8080/app1/Modern

要使用 Stopper 类在 Windows 和 Linux 平台下关闭应用程序，需要在工作目录下执行如下命令：

```

java ex14.pyrmont.startup.Stopper

```

注意，在 Catalina 的实际部署中，Stopper 类提供的关闭功能被封装在 Bootstrap 类中。

## 14.6 小结

本章介绍了 Catalina 中两个重要的组件，服务器组件和服务组件。服务器组件非常有用，因为它提供了一种优雅的方式来启动 / 关闭整个 Catalina 部署。而服务组件则封装了 servlet 容器和连接器之间的关系。本章的应用程序说明了如何使用服务器组件和服务组件，以及如何使用 StandardServer 类中关闭 Catalina 引擎的机制。



## 第 15 章

# Digester 库

正如你在前几章中看到的，我们使用 Bootstrap 类来实例化连接器、servlet 容器、Wrapper 实例和其他组件，然后调用各个对象的 set 方法将它们关联起来。例如，要实例化一个连接器和一个 servlet 容器，可以使用下面的代码：

```
Connector connector = new HttpConnector();
Context context = new StandardContext();
```

然后，使用下面的代码将它们关联起来：

```
connector.setContainer(context);
```

接着，再调用各个对象的 set 方法为其设置属性。例如，可以用 Context 实例的 setPath() 方法和 setDocBase() 方法为其设置 path 属性和 docBase 属性：

```
context.setPath("/myApp");
context.setDocBase("myApp");
```

此外，可以通过实例化各种组件，并调用 Context 对象的相应 add() 方法将这些组件添加到 Context 对象中。例如，下面的代码展示了如何为 Context 对象添加一个生命周期监听器和一个载入器：

```
LifecycleListener listener = new SimpleContextConfig();
((Lifecycle) context).addLifecycleListener(listener);
Loader loader = new WebappLoader();
context.setLoader(loader);
```

在完成关联和添加组件的所有必要操作后，就可以调用连接器的 initialize() 方法和 start() 方法，以及 servlet 容器的 start() 方法来启动整个应用程序：

```
connector.initialize();
((Lifecycle) connector).start();
((Lifecycle) context).start();
```

这种配置应用程序的方法有一个明显的缺陷，即所有的配置都必须硬编码。调整组件配置或属性值都必须重新编译 Bootstrap 类。幸运的是，Tomcat 的设计者使用了一种更加优雅的配置方式，即使用一个名为 server.xml 的 XML 文档来对应用程序进行配置。server.xml 文件中的每个元素都会转换为一个 Java 对象，元素的属性会用于设置 Java 对象的属性。这样，就可以通过简单地编辑 server.xml 文件来修改 Tomcat 的配置。例如，server.xml 文件中的 Context 元素表示一个 Context 实例：

```
<context/>
```

要为 Context 实例设置 path 属性和 docBase 属性, 可以使用这样的配置:

```
<context docBase="myApp" path="/myApp"/>
```

Tomcat 使用了开源库 Digester 来将 XML 文档中的元素转换成 Java 对象。Digester 库将在 15.1 节中介绍。

15.1 节将介绍如何配置 Web 应用程序。由于一个 Context 实例表示一个 Web 应用程序, 因此配置 Web 应用程序是通过对已经实例化的 Context 实例进行配置完成的。用来配置 Web 应用程序的 XML 文件的名称是 web.xml, 该文件位于 Web 应用程序的 WEB-INF 目录下。

## 15.1 Digester 库

Digester 是 Apache 软件基金会的 Jakarta 项目下的子 Commons 项目下的一个开源项目, 可以从 <http://jakarta.apache.org/commons/digester/> 下载 Digester 库。Digester API 包含 3 个包, 三者都被打包到 commons-digester.jar 文件中。

- org.apache.commons.digester: 该包提供了基于规则的、可处理任意 XML 文档的类;
- org.apache.commons.digester.rss: 该包包含了一些可以用来解析与很多新闻源使用的 RSS (Rich Site Summary, 富站点摘要) 格式兼容的 XML 文档的例子;
- org.apache.commons.digester.xmlrules: 该包为 Digester 库提供了一些规则基于 XML 的定义。

我们并不会介绍这 3 个包中的所有成员, 相反, 会主要介绍 Tomcat 使用到的一些重要的类。首先, 我们会介绍一下 Digester 库中最重要的一类——Digester 类。

### 15.1.1 Digester 类

org.apache.commons.digester.Digester 类是 Digester 库中的主类。该类可用于解析 XML 文档。对于 XML 文档中的每个元素, Digester 对象都会检查它是否要做事先预定义的事件。在调用 Digester 对象的 parse() 方法之前, 程序员要先定义好 Digester 对象执行哪些动作。

那么如何定义在 Digester 对象遇到某个 XML 元素时它应该执行什么动作呢? 很简单。程序员要先定义好模式, 然后将每个模式与一条或多条规则相关联。XML 文档中根元素的模式与元素的名字相同。例如, 考虑代码清单 15-1 中的 XML 文档:

代码清单 15-1 example.xml 文件

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<employee firstName="Brian" lastName="May">
  <office>
    <address streeName="Wellington Street" streetNumber="110"/>
  </office>
</employee>
```

该 XML 文档中的根元素是 employee。employee 元素有一个模式, 名为 employee。office 元素是 employee 元素的子元素。子元素的模式是由该元素的父元素的模式再加上 "/" 符号, 以及该元素名字拼接而成的。因此, office 元素的模式是 employee/office。以此类推, address 元素的

模式由如下字符串拼接而成：

父元素的模式 + “/” + 该元素的名字

其中，address 元素的父元素是 office 元素，office 元素的模式是 “employee/office”，因此，address 元素的模式是 employee/office/address。

既然你应该已经理解了如何从 XML 文档中推导出元素的模式。下面就讨论一下规则。

一条规则指明了当 Digester 对象遇到了某个特殊的模式时要执行的一个或多个动作。规则是 org.apache.commons.digester.Rule 类的实例。Digester 类可以包含 0 个或多个 Rule 对象。在 Digester 实例中，这些规则和其相关联的模式都存储在由 org.apache.commons.digester.Rules 接口表示的一类存储器中。每当把一条规则添加到 Digester 实例中时，Rule 对象都会被添加到 Rules 对象中。

另外，Rule 类有 begin() 方法和 end() 方法。在解析 XML 文档时，当 Digester 实例遇到匹配某个模式的元素的开始标签时，它会调用相应的 Rule 对象的 begin() 方法。而当 Digester 实例遇到相应元素的结束标签时，它会调用 Rule 对象的 end() 方法。

当解析代码清单 15-1 中的 example.xml 文档时，Digester 对象会执行下面的操作：

- 1) Digester 对象首先会遇到 employee 元素的开始标签，因此，它会检查是否有规则和模式 employee 相关联。若有，Digester 会执行相应 Rule 对象的 begin() 方法；若有多个 Rule 对象与该模式相匹配，则按照其添加到 Digester 对象的顺序逐个执行；
- 2) 然后，它会遇到 office 元素的开始标签，Digester 对象会检查是否有规则与模式 employee/office 相关联，若有，则它执行 Rule 对象实现的 begin() 方法；
- 3) 接下来，Digester 实例遇到 address 元素的开始标签，它会检查是否有规则与模式 employee/office/address 相关联，若有，则它调用相应 Rule 对象的 begin() 方法；
- 4) 接着，Digester 对象会遇到 address 元素的结束标签，会执行相匹配的 Rule 对象的 end() 方法；
- 5) 再下来，Digester 对象会遇到 office 元素的结束标签，会执行相匹配的 Rule 对象的 end() 方法；
- 6) 最后，Digester 对象会遇到 employee 元素的结束标签，会执行相匹配的 Rule 对象的 end() 方法。

那么有哪些规则可以使用呢？Digester 库已经预定义了一些规则。可以直接使用这些规则，而无须深入理解 Rule 类的实现。但是，如果这些规则仍不够，你可以实现自己的规则。预定义的规则包括创建对象和设置属性值等的规则。

#### 1. 创建对象

若想要 Digester 对象在遇到某个特殊的模式时创建对象，则需要调用其 addObjectCreate() 方法。该方法有 4 个重载版本。下面是两个比较常用的重载方法的签名：

```
public void addObjectCreate(java.lang.String pattern, java.lang.Class clazz)
public void addObjectCreate(java.lang.String pattern, java.lang.String className)
```



需要传入一个模式和一个 Class 对象或类名来调用该方法。例如，如果你想让 Digester 对象在遇到模式 `employee` 时，创建一个 `ex15.pyrmont.digester.test.Employee` 对象，则可以用下面的代码来调用 `addObjectCreate()` 方法：

```
digester.addObjectCreate("employee",  
    ex15.pyrmont.digester.test.Employee.class);
```

或

```
digester.addObjectCreate("employee",  
    "ex15.pyrmont.digester.test.Employee");
```

`addObjectCreate()` 方法的另外两个重载版本允许在 XML 文档中定义类的名字，而无须将其作为参数传入。这一点非常有用，因为这使得类名可以在运行时决定。下面是这两个重载方法的签名：

```
public void addObjectCreate(java.lang.String pattern,  
    java.lang.String className, java.lang.String attributeName)
```

```
public void addObjectCreate(java.lang.String pattern,  
    java.lang.String attributeName, java.lang.Class clazz)
```

在这两个重载方法中，参数 `attributeName` 参数指明了 XML 元素的属性的名字，该属性包含了将要实例化的类的名字。例如，可以使用下面的代码来添加创建对象的一条规则：

```
digester.addObjectCreate("employee", null, "className");
```

其中，XML 元素的属性名为 `className`。

然后，可以传入 XML 元素中的类名：

```
<employee firstName="Brian" lastName="May"  
    className="ex15.pyrmont.digester.test.Employee">
```

或者，可以在 `addObjectCreate()` 方法中定义默认类名：

```
digester.addObjectCreate("employee",  
    "ex15.pyrmont.digester.test.Employee", "className");
```

如果 `employee` 元素包含 `className` 属性，那么该属性指定的值会用来作为待实例化的类的名字。如果没有 `className` 属性，则会使用默认类名。

`addObjectCreate()` 方法创建的对象会被压入到一个内部栈中。Digester 库提供了一些其他的方法来对新建的对象执行查看、入栈、出栈等操作。

## 2. 设置属性

另一个比较有用的方法是 `addSetProperties()` 方法，使用该方法可以使 Digester 对象为创建的对象设置属性。该方法的一个重载版本的方法签名是：

```
public void addSetProperties(java.lang.String pattern)
```

使用时，传入一个模式到该方法中。例如，考虑下面的代码：

```
digester.addObjectCreate("employee",  
    "ex15.pyrmont.digester.test.Employee");  
digester.addSetProperties("employee");
```

上面的 Digester 实例有两个 Rule 对象，分别用来创建对象和设置属性。它们都是通过“employee”模式触发的。Rule 对象按照其添加到 Digester 实例中的顺序逐个执行。对于下面 XML 文档中的 employee 元素（该元素匹配 employee 模式）：

```
<employee firstName="Brian" lastName="May">
```

Digester 实例首先会创建 `ex15.pyrmont.digester.test.Employee` 类的一个实例，因为创建对象的规则是第 1 个添加到 Digester 实例中。然后，Digester 实例会应用与模式 employee 相关联的第 2 条规则，调用已经实例化的 Employee 对象的 `setFirstName` 属性和 `setLastName` 属性，并分别传入参数值 Brian 和 May 来设置属性。employee 元素中的属性对应于 Employee 对象中的属性。如果 Employee 类中没有定义这样的属性会发生错误。

### 3. 调用方法

Digester 类允许通过添加一条规则，使 Digester 在遇到与该规则相关联的模式时调用内部栈最顶端对象的某个方法。这需要用到 Digester 类的 `addCallMethod()` 方法。该方法的一个重载版本的签名如下所示：

```
public void addCallMethod(java.lang.String pattern,
    java.lang.String methodName)
```

### 4. 创建对象之间的关系

Digester 实例有一个内部栈，用于临时存储创建的对象。当使用 `addObjectCreate()` 方法实例化一个类时，会把结果压入这个栈中。可以将栈想象成一口井。将对象压入栈中好像是将一个与井的直径相同的圆形物体丢入井中。从栈中弹出一个对象就像是取出最上层的对象。

当调用了两次 `addObjectCreate()` 方法时，第 1 个对象会先被丢入井中，然后是第 2 个对象。`addSetNext()` 方法会调用第 1 个对象的指定方法并将第 2 个对象作为参数传入该方法来创建第 1 个对象和第 2 个对象的关系。下面是 `addSetNext()` 方法的签名：

```
public void addSetNext(java.lang.String pattern,
    java.lang.String methodName)
```

参数 `pattern` 指明了触发该规则的具体模式，参数 `methodName` 是即将调用的第 1 个对象的方法的名称。模式应该具有如下的格式：

```
firstObject/secondObject
```

例如，employee 元素包含一个 office 元素。为了创建 employee 对象和其 office 对象之间的关系，要先调用两次 `addObjectCreate()` 方法创建对象：

```
digester.addObjectCreate("employee",
    "ex15.pyrmont.digester.test.Employee");
digester.addObjectCreate("employee/office",
    "ex15.pyrmont.digester.test.Office");
```

第 1 个 `addObjectCreate()` 方法会根据 employee 元素创建 Employee 实例。第 2 个 `addObjectCreate()` 方法在遇到 employee 元素下的 office 元素时，会创建 Office 实例。

两次调用 `addObjectCreate()` 方法创建的对象都会被压入到内部栈中。现在，Employee 对象在栈底，Office 实例在栈顶。为了创建它们之间的关系，需要另外定义一条规则，使用

addSetNext() 方法来建立关系:

```
digester.addSetNext("employee/office", "addOffice");
```

其中, addOffice() 方法定义于 Employee 类中。该方法接受一个 Office 对象作为参数。本书的第 2 个 Digester 库示例会说明如何使用 addSetNext() 方法。

### 5. 验证 XML 文档

Digester 对象解析的 XML 文档的有效性可通过某个模式进行验证。Digester 类的 validating 属性指明了是否要对 XML 文档进行有效性验证。默认情况下, 该属性的值为 false。

setValidating() 方法用来设置是否要对 XML 文档进行有效性验证。该方法的签名如下:

```
public void setValidating(boolean validating)
```

如果你想要对格式良好的 XML 文档进行有效性验证, 可以在调用该方法时传入 true。

## 15.1.2 Digester 库示例 1

本章的第 1 个示例应用程序展示了如何使用 Digester 库动态地创建对象, 并设置相应的属性值。代码清单 15-2 中的 Employee 类用做实例类, Digester 库将会创建 Employee 类的实例。

代码清单 15-2 Employee 类的定义

```
package ex15.pyrmont.digesterTest;

import java.util.ArrayList;

public class Employee {
    private String firstName;
    private String lastName;
    private ArrayList offices = new ArrayList();

    public Employee() {
        System.out.println("Creating Employee");
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        System.out.println("Setting firstName : " + firstName);
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        System.out.println("Setting lastName : " + lastName);
        this.lastName = lastName;
    }

    public void addOffice(Office office) {
        System.out.println("Adding Office to this employee");
        offices.add(office);
    }

    public ArrayList getOffices() {
        return offices;
    }
}
```



```

public void printName() {
    System.out.println("My name is " + firstName + " " + lastName);
}
}

```

Employee 类有 3 个属性，分别是 firstName、lastName 和 office。属性 firstName 和 lastName 是字符串类型的，属性 office 是 ex15.pyrmont.digester.Office 类型的。office 属性将会在 Digester 库的第 2 个示例中使用。

Employee 类还有一个方法，printName()。该方法仅仅是将属性 firstName 和 lastName 值输出到控制台上。

现在，要写一个测试类 Test01，它使用 Digester 类，并为其添加创建 Employee 对象和设置其属性的规则。代码清单 15-3 给出了 Test01 类的定义。

代码清单 15-3 Test01 类的定义

```

package ex15.pyrmont.digestertest;

import java.io.File;
import org.apache.commons.digester.Digester;

public class Test01 {

    public static void main(String[] args) {
        String path = System.getProperty("user.dir") + File.separator +
            "etc";
        File file = new File(path, "employee.xml");
        Digester digester = new Digester();
        // add rules
        digester.addObjectCreate("employee",
            "ex15.pyrmont.digestertest.Employee");
        digester.addSetProperties("employee");
        digester.addCallMethod("employee", "printName");

        try {
            Employee employee = (Employee) digester.parse(file);
            System.out.println("First name : " + employee.getFirstName());
            System.out.println("Last name : " + employee.getLastName());
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

上述代码首先定义了包含 XML 文档位置的路径，并将其传入到 File 类的构造函数中。然后，使用下面的代码创建一个 Digester 对象，并为模式 employee 添加 3 条规则：

```

digester.addObjectCreate("employee",
    "ex15.pyrmont.digestertest.Employee");
digester.addSetProperties("employee");
digester.addCallMethod("employee", "printName");

```

接下来，调用 Digester 对象的 parse() 方法，并传入引用 XML 文档的 File 对象作为参数。parse() 方法的返回值是 Digester 对象的内部栈中的第 1 个对象：

```
Employee employee = (Employee) digester.parse(file);
```

这样，就获得了 Digester 对象实例化的 Employee 对象。若要查看是否设置了 Employee 对象的属性，可以调用 Employee 对象的 `getFirstName()` 方法和 `getLastName()` 方法：

```
System.out.println("First name : " + employee.getFirstName());  
System.out.println("Last name : " + employee.getLastName());
```

现在，代码清单 15-4 给出了 `employee1.xml` 文档的内容，根元素为 `employee`。该元素有两个属性，分别是 `firstName` 和 `lastName`：

代码清单 15-4 `employee1.xml` 文件的内容

```
<?xml version="1.0" encoding="ISO-8859-1"?>  
<employee firstName="Brian" lastName="May">  
</employee>
```

运行 Test01 类的输出结果如下所示：

```
Creating Employee  
Setting firstName : Brian  
Setting lastName : May  
My name is Brian May  
First name : Brian  
Last name : May
```

这就是执行的操作。

当调用 Digester 对象的 `parse()` 方法时，它会打开指定的 XML 文档，开始解析它。首先，Digester 类会查看 `employee` 元素的开始标签。这会触发与 `employee` 模式关联的 3 条规则，按照其被添加到 Digester 对象中的顺序逐个执行。第 1 条规则用于创建 Employee 对象。因此，Digester 对象会实例化 Employee 类，调用 Employee 类的构造函数。Employee 类的构造函数会输出字符串 “Creating Employee”。

第 2 条规则设置 Employee 对象的属性。在 `employee` 元素中包含两个属性，分别是 `firstName` 和 `lastName`。该规则会调用属性 `firstName` 和 `lastName` 的 `set` 方法，分别输出以下字符串：

```
Setting firstName : Brian  
Setting lastName : May
```

第 3 条规则会调用 Employee 类的 `printName()` 方法，输出如下的字符串：

```
My name is Brian May
```

最后两行是调用 Employee 对象的 `getFirstName()` 方法和 `getLastName()` 方法的输出内容：

```
First name : Brian  
Last name : May
```

### 15.1.3 Digester 库示例 2

本章的第 2 个示例 Digester 说明了如何创建两个对象，并建立他们之间的关系。首先要定义要创建的关系的类型。例如，员工可以工作在一个或多个办公室中。办公室由 Office 类的实例表示。可以创建一个 Employee 对象和一个 Office 对象，并为它们创建关系。代码清单 15-5 给出了 Office 类的定义。

代码清单 15-5 Office 类的定义

```

package ex15.pyrmont.digesterestest;

public class Office {
    private Address address;
    private String description;
    public Office() {
        System.out.println("..Creating Office");
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        System.out.println("..Setting office description : " +
description);
        this.description = description;
    }
    public Address getAddress() {
        return address;
    }
    public void setAddress(Address address) {
        System.out.println("..Setting office address : " + address);
        this.address = address;
    }
}

```

可以通过调用父对象的某个方法来创建关系。注意，本示例使用代码清单 15-2 中的 Employee 类。Employee 类有一个 addOffice() 方法，将一个 Office 对象添加到其集合类型的变量 offices 中。

如果没有 Digester 库，则需要编写如下的代码：

```

Employee employee = new Employee();
Office office = new Office();
employee.addOffice(office);

```

一间办公室有一个地址，地址是由 Address 类的实例表示的。代码清单 15-6 给出了 Address 类的定义。

代码清单 15-6 Address 类的定义

```

package ex15.pyrmont.digesterestest;

public class Address {
    private String streetName;
    private String streetNumber;
    public Address() {
        System.out.println("....Creating Address");
    }
    public String getStreetName() {
        return streetName;
    }
    public void setStreetName(String streetName) {
        System.out.println("....Setting streetName : " + streetName);
        this.streetName = streetName;
    }
    public String getStreetNumber() {
        return streetNumber;
    }
}

```



```

public void setStreetNumber(String streetNumber) {
    System.out.println("....Setting streetNumber : " + streetNumber);
    this.streetNumber = streetNumber;
}
public String toString() {
    return "...." + streetNumber + " " + streetName;
}
}

```

若要将 Address 对象关联到 Office 对象中，则可以调用 Office 类的 setAddress() 方法。若没有 Digester 库的帮助，则需要编写如下的代码：

```

Office office = new Office();
Address address = new Address();
office.setAddress(address);

```

关于 Digester 库的第 2 个示例应用程序展示了如何使用 Digester 库创建对象并为其创建关系。这里将要使用到的类有 Employee 类、Office 类和 Address 类。代码清单 15-7 给出了 Test02 类的定义，该类使用一个 Digester 对象，并为其添加规则。

代码清单 15-7 Test02 类的定义

```

package ex15.pyrmont.digestertest;

import java.io.File;
import java.util.*;
import org.apache.commons.digester.Digester;

public class Test02 {

    public static void main(String[] args) {
        String path = System.getProperty("user.dir") + File.separator +
            "etc";
        File file = new File(path, "employee2.xml");
        Digester digester = new Digester();
        // add rules
        digester.addObjectCreate("employee",
            "ex15.pyrmont.digestertest.Employee");
        digester.addSetProperties("employee");
        digester.addObjectCreate("employee/office",
            "ex15.pyrmont.digestertest.Office");
        digester.addSetProperties("employee/office");
        digester.addSetNext("employee/office", "addOffice");
        digester.addObjectCreate("employee/office/address",
            "ex15.pyrmont.digestertest.Address");
        digester.addSetProperties("employee/office/address");
        digester.addSetNext("employee/office/address", "setAddress");
        try {
            Employee employee = (Employee) digester.parse(file);
            ArrayList offices = employee.getOffices();
            Iterator iterator = offices.iterator();
            System.out.println(
                "-----");
            while (iterator.hasNext()) {
                Office office = (Office) iterator.next();
                Address address = office.getAddress();
                System.out.println(office.getDescription());
                System.out.println("Address : " +

```

```

        address.getStreetNumber() + " " + address.getStreetName());
        System.out.println("-----");
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

要想知道 Digester 对象所执行的动作，可以查看代码清单 15-8 中给出的 XML 文档 employee2.xml 的内容。

代码清单 15-8 employee2.xml 文件的内容

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<employee firstName="Freddie" lastName="Mercury">
  <office description="Headquarters">
    <address streetName="Wellington Avenue" streetNumber="223"/>
  </office>
  <office description="Client site">
    <address streetName="Downing Street" streetNumber="10"/>
  </office>
</employee>

```

运行 Test02 类会产生如下的输出：

```

Creating Employee
Setting firstName : Freddie
Setting lastName : Mercury
..Creating Office
..Setting office description : Headquarters
....Creating Address
....Setting streetName : Wellington Avenue
....Setting streetNumber : 223
..Setting office address : ....223 Wellington Avenue
Adding Office to this employee
..Creating Office
..Setting office description : Client site
....Creating Address
....Setting streetName : Downing Street
....Setting streetNumber : 10
..Setting office address : ....10 Downing Street
Adding Office to this employee
-----
Headquarters
Address : 223 Wellington Avenue
-----
Client site
Address : 10 Downing Street
-----

```

#### 15.1.4 Rule 类

Rule 类包含一些方法，其中最重要的两个方法是 begin() 方法和 end() 方法。当 Digester 实例遇到某个 XML 元素的开始标签时，它会调用它所包含的匹配 Rule 对象的 begin() 方法。Rule 类的 begin() 方法的签名如下：

```
public void begin(org.xml.sax.Attributes attributes)
    throws java.lang.Exception
```

当 Digester 实例遇到某个 XML 元素的结束标签时，它会调用它所包含的匹配所有 Rule 实例的 end() 方法。Rule 类的 end() 方法的签名如下：

```
public void end() throws java.lang.Exception
```

Digester 对象是如何完成这些工作的呢？当调用 Digester 对象的 addObjectCreate() 方法、addCallMethod() 方法、addSetNext() 方法或其他方法时，都会间接地调用 Digester 类的 addRule() 方法。该方法会将一个 Rule 对象和它所匹配的模式添加到 Digester 对象的 Rules 集合中。

addRule() 方法的签名如下：

```
public void addRule(java.lang.String pattern, Rule rule)
```

Digester 类的 addRule() 方法的实现如下：

```
public void addRule(String pattern, Rule rule) {
    rule.setDigester(this);
    getRules().add(pattern, rule);
}
```

查看 Digester 类的 addObjectCreate() 方法的重载实现如下：

```
public void addObjectCreate(String pattern, String className) {
    addRule(pattern, new ObjectCreateRule(className));
}
public void addObjectCreate(String pattern, Class clazz) {
    addRule(pattern, new ObjectCreateRule(clazz));
}
public void addObjectCreate(String pattern, String className,
    String attributeName) {
    addRule(pattern, new ObjectCreateRule(className, attributeName));
}
public void addObjectCreate(String pattern,
    String attributeName, Class clazz) {
    addRule(pattern, new ObjectCreateRule(attributeName, clazz));
}
```

这 4 个重载的方法都调用了 addRule() 方法，ObjectCreateRule 类是 Rule 的子类，该类的实例可作为 addRule() 方法的第二个参数使用。下面是 ObjectCreateRule 类中 begin() 方法和 end() 方法的实现：

```
public void begin(Attributes attributes) throws Exception {
    // Identify the name of the class to instantiate
    String realClassName = className;
    if (attributeName != null) {
        String value = attributes.getValue(attributeName);
        if (value != null) {
            realClassName = value;
        }
    }
    if (digester.log.isDebugEnabled()) {
        digester.log.debug("[ObjectCreateRule]{" + digester.match +
            "}New " + realClassName);
    }
}
```



```

// Instantiate the new object and push it on the context stack
Class clazz = digester.getClassLoader().loadClass(realClassName);
Object instance = clazz.newInstance();
digester.push(instance);
}

public void end() throws Exception {
    Object top = digester.pop();
    if (digester.log.isDebugEnabled()) {
        digester.log.debug("[ObjectCreateRule]" + digester.match +
            " Pop " + top.getClass().getName());
    }
}

```

begin() 方法的最后 3 行会创建 Digester 对象的一个实例，并将其压入到 Digester 对象的内部栈中。end() 方法会将内部栈的栈顶元素弹出栈。

Rule 类的其他子类具有相类似的功能。如果你想知道每条规则的内幕，可以打开源代码。

### 15.1.5 Digester 库示例 3：使用 RuleSet

要向 Digester 实例中添加 Rule 对象，还可以调用其 addRuleSet() 方法，方法签名如下：

```
public void addRuleSet(RuleSet ruleSet)
```

Rule 对象集合是 org.apache.commons.digester.RuleSet 接口的实例。该接口定义了两个方法，分别是 addRuleInstance() 和 getNamespaceURI()。addRuleInstance() 方法的签名如下：

```
public void addRuleInstance(Digester digester)
```

addRuleInstance() 方法将在当前 RuleSet 中的 Rule 对象的集合作为该方法的参数添加到 Digester 实例中。

getNamespaceURI() 方法返回将要应用在 RuleSet 中所有 Rule 对象的命名空间的 URI。该方法签名如下：

```
public java.lang.String getNamespaceURI()
```

因此，在创建了 Digester 对象之后，可以创建一个 RuleSet 对象，并调用 addRuleSet() 方法将其添加到 Digester 对象中。

实现 RuleSet 接口有一个基类 RuleSetBase，方便使用。RuleSetBase 类是一个抽象类，提供了 getNamespaceURI() 方法的实现。你只需要提供 addRuleInstances() 方法的实现就可以了。

作为一个例子，这里会修改前面例子中使用过的 Test02 类，引入 EmployeeRuleSet 类。代码清单 15-9 给出了 EmployeeRuleSet 类的定义。

代码清单 15-9 EmployeeRuleSet 类的定义

```

package ex15.pyrmont.digestertest;

import org.apache.commons.digester.Digester;
import org.apache.commons.digester.RuleSetBase;

public class EmployeeRuleSet extends RuleSetBase {

```

```

public void addRuleInstances(Digester digester) {
    // add rules
    digester.addObjectCreate("employee",
        "ex15.pyrmont.digestertest.Employee");
    digester.addSetProperties("employee");
    digester.addObjectCreate("employee/office",
        "ex15.pyrmont.digestertest.Office");
    digester.addSetProperties("employee/office");
    digester.addSetNext("employee/office", "addOffice");
    digester.addObjectCreate("employee/office/address",
        "ex15.pyrmont.digestertest.Address");
    digester.addSetProperties("employee/office/address");
    digester.addSetNext("employee/office/address", "setAddress");
}
}

```

注意, `EmployeeRuleSet` 类中的 `addRuleInstances()` 方法的实现的功能像 `Test02` 类中一样, 将相同的 `Rule` 对象添加到 `Digester` 对象中。代码清单 15-10 给出了 `Test03` 类的定义。`Test03` 类中会创建 `EmployeeRuleSet` 类的一个实例, 然后将其添加中之前创建的 `Digester` 对象中。

代码清单 15-10 `Test03` 的定义

```

package ex15.pyrmont.digestertest;

import java.io.File;
import java.util.ArrayList;
import java.util.Iterator;
import org.apache.commons.digester.Digester;

public class Test03 {
    public static void main(String[] args) {
        String path = System.getProperty("user.dir") +
            File.separator + "etc";
        File file = new File(path, "employee2.xml");
        Digester digester = new Digester();
        digester.addRuleSet(new EmployeeRuleSet());
        try {
            Employee employee = (Employee) digester.parse(file);
            ArrayList offices = employee.getOffices();
            Iterator iterator = offices.iterator();
            System.out.println(
                "-----");
            while (iterator.hasNext()) {
                Office office = (Office) iterator.next();
                Address address = office.getAddress();
                System.out.println(office.getDescription());
                System.out.println("Address : " +
                    address.getStreetNumber() + " " + address.getStreetName());
                System.out.println("-----");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

运行 Test03 类会产生和 Test02 类相同的输出。但是请注意，Test03 类的代码量更少一些，因为添加 Rule 对象的代码隐藏在 EmployeeRuleSet 类中。

正如你将在接下来的几节中看到的，Catalina 使用了 RuleSetBase 类的子类来初始化它的服务器组件和其他的组件。在下面的小节中，你会看到 Digester 库在 Catalina 中所扮演的重要角色。

## 15.2 ContextConfig 类

与其他类型的容器不同，StandardContext 实例必须有一个监听器，这个监听器会负责配置 StandardContext 实例，设置成功后会将 StandardContext 实例的变量 configured 置为 true。在前面的章节中，我们使用 SimpleContextConfig 类的实例作为 StandardContext 实例的监听器，这个类非常简单，其唯一的用途就是设置 configured 变量，这样 StandardContext 类的 start() 方法才能继续执行。

在 Tomcat 的实际部署中，StandardContext 类的标准监听器是 org.apache.catalina.startup.ContextConfig 类的一个实例。与我们自己实现的简陋的 SimpleContextConfig 类不同，ContextConfig 会执行很多对 StandardContext 实例来说必不可少的任务。例如，与某个 StandardContext 实例关联的 ContextConfig 实例会安装一个验证器阀到 StandardContext 实例的管道对象中。此外，它还会添加一个许可阀（org.apache.catalina.valves.CertificateValve 类的实例）到管道对象中。

但更重要的是，ContextConfig 类的实例还要读取和解析默认的 web.xml 文件和应用程序自定义的 web.xml 文件，并将 XML 元素转换为 Java 对象。默认的 web.xml 文件位于 CATALINE\_HOME 目录下的 conf 目录中。其中定义并映射了很多默认的 servlet，配置了很多 MIME 类型文件的映射，定义了默认的 Session 超时时间，以及定义了欢迎文件的列表。你可以找到该文件并阅读一下其中的内容。

应用程序的 web.xml 文件是应用程序自定义的配置文件，位于应用程序目录下的 WEB-INF 目录中。这两个文件都不是必需的，即使这两个文件都没有找到，ContextConfig 实例仍然会继续执行。

ContextConfig 实例会创建为每个 servlet 元素创建一个 StandardWrapper 类。因此，正如你在本章应用程序中看到的，配置变简单了。不再需要实例化一个 Wrapper 实例了。

因此，在 Bootstrap 类中，需要实例化一个 ContextConfig 类，并调用 org.apache.catalina.Lifecycle 接口的 addLifecycleListener() 方法将 ContextConfig 实例添加到 StandardContext 实例中：

```
LifecycleListener listener = new ContextConfig();
((Lifecycle) context).addLifecycleListener(listener);
```

在启动 StandardContext 实例时，会触发以下事件：

- BEFORE\_START\_EVENT
- START\_EVENT
- AFTER\_START\_EVENT

当程序停止时，StandardContext 实例会触发以下事件：

- BEFORE\_STOP\_EVENT



- STOP\_EVENT
- AFTER\_STOP\_EVENT

ContextConfig 实例会对两种事件做出相应，分别是 START\_EVENT 和 STOP\_EVENT。每次 StandardContext 实例触发事件时，会调用 ContextConfig 实例的 lifecycleEvent() 方法。代码清单 15-11 给出了 lifecycleEvent() 方法的实现。在代码清单 15-11 中添加了注释以便 stop() 方法更易于理解。

代码清单 15-11 ContextConfig 类的 lifecycleEvent() 方法

```
public void lifecycleEvent(LifecycleEvent event) {
    // Identify the context we are associated with
    try {
        context = (Context) event.getLifecycle();
        if (context instanceof StandardContext) {
            int contextDebug = ((StandardContext) context).getDebug();
            if (contextDebug > this.debug)
                this.debug = contextDebug;
        }
    }
    catch (ClassCastException e) {
        log(sm.getString("contextConfig.cce", event.getLifecycle(), e));
        return;
    }
    // Process the event that has occurred
    if (event.getType().equals(Lifecycle.START_EVENT))
        start();
    else if (event.getType().equals(Lifecycle.STOP_EVENT))
        stop();
}
```

正如你在 lifecycleEvent() 方法末尾看到的，它会调用其 start() 方法或 stop() 方法。代码清单 15-12 给出了 start() 方法的实现。注意，在其主体中 start() 方法会调用 defaultConfig() 方法和 applicationConfig() 方法。这两个方法将在后面几节中介绍。

代码清单 15-12 ContextConfig 类的 start() 方法的实现

```
private synchronized void start() {
    if (debug > 0)
        log(sm.getString("contextConfig.start"));
    // reset the configured boolean
    context.setConfigured(false);
    // a flag that indicates whether the process is still
    // going smoothly
    ok = true;
    // Set properties based on DefaultContext
    Container container = context.getParent();
    if (!context.getOverride()) {
        if (container instanceof Host) {
            ((Host) container).importDefaultContext(context);
            container = container.getParent();
        }
        if (container instanceof Engine) {
            ((Engine) container).importDefaultContext(context);
        }
    }
}
```

```

// Process the default and application web.xml files
defaultConfig();
applicationConfig();
if (ok) {
    validateSecurityRoles();
}
// Scan tag library descriptor files for additional listener classes
if (ok) {
    try {
        tldScan();
    }
    catch (Exception e) {
        log(e.getMessage(), e);
        ok = false;
    }
}
// Configure a certificates exposier valve, if required
if (ok)
    certificatesConfig();

// Configure an authenticator if we need one
if (ok)
    authenticatorConfig();
// Dump the contents of this pipeline if requested
if ((debug >= 1) && (context instanceof ContainerBase)) {
    log("Pipeline Configuration:");
    Pipeline pipeline = ((ContainerBase) context).getPipeline();
    Valve valves[] = null;
    if (pipeline != null)
        valves = pipeline.getValves();
    if (valves != null) {
        for (int i = 0; i < valves.length; i++) {
            log(" " + valves[i].getInfo());
        }
    }
    log("=====");
}
// Make our application available if no problems were encountered
if (ok)
    context.setConfigured(true);
else {
    log(sm.getString("contextConfig.unavailable"));
    context.setConfigured(false);
}
}
}

```

### 15.2.1 defaultConfig() 方法

defaultConfig() 方法负责读取并解析位于 %CATALINA\_HOME%/conf 目录下的默认 web.xml 文件。代码清单 15-13 给出了 defaultConfig() 方法的实现。

代码清单 15-13 defaultConfig() 方法的实现

```

private void defaultConfig() {
    // Open the default web.xml file, if it exists
    File file = new File(Constants.DefaultWebXml);
    if (!file.isAbsolute())

```

```

file = new File(System.getProperty("catalina.base"),
    Constants.DefaultWebXml);
FileInputStream stream = null;
try {
    stream = new FileInputStream(file.getCanonicalPath());
    stream.close();
    stream = null;
}
catch (FileNotFoundException e) {
    log(sm.getString("contextConfig.defaultMissing"));
    return;
}
catch (IOException e) {
    log(sm.getString("contextConfig.defaultMissing"), e);
    return;
}
// Process the default web.xml file
synchronized (webDigester) {
    try {
        InputSource is =
            new InputSource("file://" + file.getAbsolutePath());
        stream = new FileInputStream(file);
        is.setByteStream(stream);
        webDigester.setDebug(getDebug());
        if (context instanceof StandardContext)
            ((StandardContext) context).setReplaceWelcomeFiles(true);
        webDigester.clear();
        webDigester.push(context);
        webDigester.parse(is);
    }
    catch (SAXParseException e) {
        log(sm.getString("contextConfig.defaultParse"), e);
        log(sm.getString("contextConfig.defaultPosition",
            "" + e.getLineNumber(), "" + e.getColumnNumber()));
        ok = false;
    }
    catch (Exception e) {
        log(sm.getString("contextConfig.defaultParse"), e);
        ok = false;
    }
    finally {
        try {
            if (stream != null) {
                stream.close();
            }
        }
        catch (IOException e) {
            log(sm.getString("contextConfig.defaultClose"), e);
        }
    }
}
}

```

defaultConfig() 方法首先会创建一个 File 对象，该 File 对象引用默认的 web.xml 配置文件：

```
File file = new File(Constants.DefaultWebXml);
```

常量 DefaultWebXML 的值定义在 org.apache.catalina.startup.Constants 类中：

```
public static final String DefaultWebXml = "conf/web.xml";
```



然后, `defaultConfig()` 方法处理 `web.xml` 文件。它会锁定 `webDigester` 对象变量, 然后开始解析该文件:

```
synchronized (webDigester) {
    try {
        InputSource is =
            new InputSource("file://" + file.getAbsolutePath());
        stream = new FileInputStream(file);
        is.setByteStream(stream);
        webDigester.setDebug(getDebug());
        if (context instanceof StandardContext)
            ((StandardContext) context).setReplaceWelcomeFiles(true);
        webDigester.clear();
        webDigester.push(context);
        webDigester.parse(is);
    }
```

对象变量 `webDigester` 是一个指向 `Digester` 实例的引用, 该 `Digester` 实例已经包含了处理 `web.xml` 文件所要使用的规则。这将在 15.2.3 节中讨论。

### 15.2.2 applicationConfig() 方法

`applicationConfig()` 方法与 `defaultConfig()` 方法类似, 只不过它处理的是应用程序自定义的部署描述符, 该部署描述符位于应用程序目录下的 `WEB-INF` 目录中。

代码清单 15-14 给出了 `applicationConfig()` 方法的实现。

代码清单 15-14 applicationConfig() 方法的实现

---

```
private void applicationConfig() {
    // Open the application web.xml file, if it exists
    InputStream stream = null;
    ServletContext servletContext = context.getServletContext();
    if (servletContext != null)
        stream = servletContext.getResourceAsStream
            (Constants.ApplicationWebXml);
    if (stream == null) {
        log(sm.getString("contextConfig.applicationMissing"));
        return;
    }

    // Process the application web.xml file
    synchronized (webDigester) {
        try {
            URL url =
                servletContext.getResource(Constants.ApplicationWebXml);

            InputSource is = new InputSource(url.toExternalForm());
            is.setByteStream(stream);
            webDigester.setDebug(getDebug());
            if (context instanceof StandardContext) {
                ((StandardContext) context).setReplaceWelcomeFiles(true);
            }
            webDigester.clear();
            webDigester.push(context);
            webDigester.parse(is);
        }
        catch (SAXParseException e) {
            log(sm.getString("contextConfig.applicationParse"), e);
            log(sm.getString("contextConfig.applicationPosition",
```

```

        "" + e.getLineNumber(),
        "" + e.getColumnNumber());
    ok = false;
}
catch (Exception e) {
    log(sm.getString("contextConfig.applicationParse"), e);
    ok = false;
}
finally {
    try {
        if (stream != null) {
            stream.close();
        }
    }
    catch (IOException e) {
        log(sm.getString("contextConfig.applicationClose"), e);
    }
}
}
}
}

```

### 15.2.3 创建 Web Digester

在 ContextConfig 类中，使用变量 webDigester 来引用一个 Digester 类型的对象：

```
private static Digester webDigester = createWebDigester();
```

这个 Digester 对象用于解析默认的 web.xml 文件和应用程序自定义的 web.xml 文件。在调用 createWebDigester() 方法时会添加用来处理 web.xml 文件的规则。代码清单 15-15 给出了 createWebDigester() 方法的实现。

代码清单 15-15 createWebDigester() 方法的实现

```

private static Digester createWebDigester() {
    URL url = null;
    Digester webDigester = new Digester();
    webDigester.setValidating(true);
    url = ContextConfig.class.getResource(
        Constants.WebDtdResourcePath_22);
    webDigester.register(Constants.WebDtdPublicId_22,
        url.toString());
    url = ContextConfig.class.getResource(
        Constants.WebDtdResourcePath_23);
    webDigester.register(Constants.WebDtdPublicId_23,
        url.toString());
    webDigester.addRuleSet(new WebRuleSet());
    return (webDigester);
}

```

注意，createWebDigester() 方法调用了变量 webDigester 的 addRuleSet() 方法，传入了一个 org.apache.catalina.startup.WebRuleSet 类型的对象作为参数。WebRuleSet 类是 org.apache.commons.digester.RuleSetBase 类的子类。如果你对 servlet 应用程序部署描述器的语法熟悉，并且已经阅读过了 15.1 节的话，你应该可以理解它的工作原理。

代码清单 15-16 给出了 WebRuleSet 类的定义。注意，为了节省空间，这里已经移除了

addRuleInstances() 方法的实现部分。

代码清单 15-16 WebRuleSet 类的定义

```
package org.apache.catalina.startup;

import java.lang.reflect.Method;
import org.apache.catalina.Context;
import org.apache.catalina.Wrapper;
import org.apache.catalina.deploy.SecurityConstraint;
import org.apache.commons.digester.Digester;
import org.apache.commons.digester.Rule;
import org.apache.commons.digester.RuleSetBase;
import org.xml.sax.Attributes;

/**
 * <p><strong>RuleSet</strong> for processing the contents of a web
 * application
 * deployment descriptor (<code>/WEB-INF/web.xml</code>) resource.</p>
 *
 * @author Craig R. McClanahan
 * @version $Revision: 1.1 $ $Date: 2001/10/17 00:44:02 $
 */
public class WebRuleSet extends RuleSetBase {
    // ----- Instance Variables
    /**
     * The matching pattern prefix to use for recognizing our elements.
     */
    protected String prefix = null;

    // ----- Constructor
    /**
     * Construct an instance of this <code>RuleSet</code> with
     * the default matching pattern prefix.
     */
    public WebRuleSet() {
        this("");
    }

    /**
     * Construct an instance of this <code>RuleSet</code> with
     * the specified matching pattern prefix.
     *
     * @param prefix Prefix for matching pattern rules (including the
     * trailing slash character)
     */
    public WebRuleSet(String prefix) {
        super();
        this.namespaceURI = null;
        this.prefix = prefix;
    }

    // ----- Public Methods
    /**
     * <p>Add the set of Rule instances defined in this RuleSet to the
     * specified <code>Digester</code> instance, associating them with
     * our namespace URI (if any). This method should only be called
     * by a Digester instance.</p>
     *
     * @param digester Digester instance to which the new Rule instances
     * should be added.
     */
}
```



```

public void addRuleInstances(Digester digester) {
    digester.addRule(prefix + "web-app",
        new SetPublicIdRule(digester, "setPublicId"));
    digester.addCallMethod(prefix + "web-app/context-param",
        "addParameter", 2);
    digester.addCallParam(prefix +
        "web-app/context-param/param-name", 0);
    digester.addCallParam(prefix +
        "web-app/context-param/param-value", 1);
    digester.addCallMethod(prefix + "web-app/display-name",
        "setDisplayName", 0);
    digester.addRule(prefix + "web-app/distributable",
        new SetDistributableRule(digester));
    ...
    digester.addObjectCreate(prefix + "web-app/filter",
        "org.apache.catalina.deploy.FilterDef");
    digester.addSetNext(prefix + "web-app/filter", "addFilterDef",
        "org.apache.catalina.deploy.FilterDef");
    digester.addCallMethod(prefix + "web-app/filter/description",
        "setDescription", 0);
    digester.addCallMethod(prefix + "web-app/filter/display-name",
        "setDisplayName", 0);
    digester.addCallMethod(prefix + "web-app/filter/filter-class",
        "setFilterClass", 0);
    digester.addCallMethod(prefix + "web-app/filter/filter-name",
        "setFilterName", 0);
    digester.addCallMethod(prefix + "web-app/filter/large-icon",
        "setLargeIcon", 0);
    digester.addCallMethod(prefix + "web-app/filter/small-icon",
        "setSmallIcon", 0);
    digester.addCallMethod(prefix + "web-app/filter/init-param",
        "addInitParameter", 2);
    digester.addCallParam(prefix +
        "web-app/filter/init-param/param-name", 0);
    digester.addCallParam(prefix +
        "web-app/filter/init-param/param-value", 1);
    digester.addObjectCreate(prefix + "web-app/filter-mapping",
        "org.apache.catalina.deploy.FilterMap");
    digester.addSetNext(prefix + "web-app/filter-mapping",
        "addFilterMap", "org.apache.catalina.deploy.FilterMap");
    digester.addCallMethod(prefix +
        "web-app/filter-mapping/filter-name", "setFilterName", 0);
    digester.addCallMethod(prefix +
        "web-app/filter-mapping/servlet-name", "setServletName", 0);
    digester.addCallMethod(prefix +
        "web-app/filter-mapping/url-pattern", "setURLPattern", 0);
    digester.addCallMethod(prefix +
        "web-app/listener/listener-class", "addApplicationListener", 0);
    ...
    digester.addRule(prefix + "web-app/servlet",
        new WrapperCreateRule(digester));
    digester.addSetNext(prefix + "web-app/servlet",
        "addChild", "org.apache.catalina.Container");
    digester.addCallMethod(prefix + "web-app/servlet/init-param",
        "addInitParameter", 2);
    digester.addCallParam(prefix +
        "web-app/servlet/init-param/param-name", 0);
    digester.addCallParam(prefix +
        "web-app/servlet/init-param/param-value", 1);
    digester.addCallMethod(prefix + "web-app/servlet/jsp-file",
        "setJspFile", 0);
    digester.addCallMethod(prefix +

```

```

    "web-app/servlet/load-on-startup", "setLoadOnStartupString", 0);
    digester.addCallMethod(prefix + "web-app/servlet/run-as/role-name", "setRunAs", 0);
    digester.addCallMethod(prefix + "web-app/servlet/security-role-ref", "addSecurityReference", 2);
    digester.addCallParam(prefix + "web-app/servlet/security-role-ref/role-link", 1);
    digester.addCallParam(prefix + "web-app/servlet/security-role-ref/role-name", 0);
    digester.addCallMethod(prefix + "web-app/servlet/servlet-class", "setServletClass", 0);
    digester.addCallMethod(prefix + "web-app/servlet/servlet-name", "setName", 0);
    digester.addCallMethod(prefix + "web-app/servlet-mapping", "addServletMapping", 2);
    digester.addCallParam(prefix + "web-app/servlet-mapping/servlet-name", 1);
    digester.addCallParam(prefix + "web-app/servlet-mapping/url-pattern", 0);
    digester.addCallMethod(prefix + "web-app/session-config/session-timeout", "setSessionTimeout", 1,
        new Class[] { Integer.TYPE });
    digester.addCallParam(prefix + "web-app/session-config/session-timeout", 0);
    digester.addCallMethod(prefix + "web-app/taglib", "addTaglib", 2);
    digester.addCallParam(prefix + "web-app/taglib/taglib-location",
1);
    digester.addCallParam(prefix + "web-app/taglib/taglib-uri", 0);
    digester.addCallMethod(prefix + "web-app/welcome-file-list/welcome-file", "addWelcomeFile", 0);
}
// ----- Private Classes

/**
 * A Rule that calls the <code>setAuthConstraint(true)</code> method of
 * the top item on the stack, which must be of type
 * <code>org.apache.catalina.deploy.SecurityConstraint</code>.
 */
final class SetAuthConstraintRule extends Rule {
    public SetAuthConstraintRule(Digester digester) {
        super(digester);
    }
    public void begin(Attributes attributes) throws Exception {
        SecurityConstraint securityConstraint =
            (SecurityConstraint) digester.peek();
        securityConstraint.setAuthConstraint(true);
        if (digester.getDebug() > 0)
            digester.log("Calling
SecurityConstraint.setAuthConstraint(true)");
    }
}

...
final class WrapperCreateRule extends Rule {
    public WrapperCreateRule(Digester digester) {
        super(digester);
    }
    public void begin(Attributes attributes) throws Exception {
        Context context =
            (Context) digester.peek(digester.getCount() - 1);

```

```

Wrapper wrapper = context.createWrapper();
digester.push(wrapper);
if (digester.getDebug() > 0)
    digester.log("new " + wrapper.getClass().getName());
}

public void end() throws Exception {
    Wrapper wrapper = (Wrapper) digester.pop();
    if (digester.getDebug() > 0)
        digester.log("pop " + wrapper.getClass().getName());
}
}

```

## 15.3 应用程序

本章的应用程序重在说明如何使用 ContextConfig 实例作为一个监听器来配置 StandardContext 对象。应用程序只包含 Bootstrap 一个类。代码清单 15-17 给出了 Bootstrap 类的定义。

代码清单 15-17 Bootstrap 类的定义

```

package ex15.pyrmont.startup;

import org.apache.catalina.Connector;
import org.apache.catalina.Container;
import org.apache.catalina.Context;
import org.apache.catalina.Host;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleListener;
import org.apache.catalina.Loader;
import org.apache.catalina.connector.http.HttpConnector;
import org.apache.catalina.core.StandardContext;
import org.apache.catalina.core.StandardHost;
import org.apache.catalina.loader.WebappLoader;
import org.apache.catalina.startup.ContextConfig;

public final class Bootstrap {

    // invoke: http://localhost:8080/app1/Modern or
    // http://localhost:8080/app2/Primitive
    // note that we don't instantiate a Wrapper here,
    // ContextConfig reads the WEB-INF/classes dir and loads all
    // servlets.
    public static void main(String[] args) {
        System.setProperty("catalina.base",
            System.getProperty("user.dir"));
        Connector connector = new HttpConnector();
        Context context = new StandardContext();
        // StandardContext's start method adds a default mapper
        context.setPath("/app1");
        context.setDocBase("app1");
        LifecycleListener listener = new ContextConfig();
        ((Lifecycle) context).addLifecycleListener(listener);
        Host host = new StandardHost();
        host.addChild(context);
        host.setName("localhost");
        host.setAppBase("webapps");
    }
}

```



```

Loader loader = new WebappLoader();
context.setLoader(loader);
connector.setContainer(host);
try {
    connector.initialize();
    ((Lifecycle) connector).start();
    ((Lifecycle) host).start();
    Container[] c = context.findChildren();
    int length = c.length;
    for (int i=0; i<length; i++) {
        Container child = c[i];
        System.out.println(child.getName());
    }
    // make the application wait until we press a key.
    System.in.read();
    ((Lifecycle) host).stop();
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

## 运行应用程序

要在 Windows 平台下运行应用程序，需要在工作目录中执行以下命令：

```

java -classpath ./lib/servlet.jar;./lib/commons-
collections.jar;./lib/commons-digester.jar;./lib/commons-
logging.jar;./lib/commons-beanutils.jar;./
ex15.pyrmont.startup.Bootstrap

```

而在 Linux 平台下，需要使用冒号来分隔不同库：

```

java -classpath ./lib/servlet.jar:./lib/commons-
collections.jar:./lib/commons-digester.jar:./lib/commons-
logging.jar:./lib/commons-beanutils.jar:./
ex15.pyrmont.startup.Bootstrap

```

要调用 PrimitiveServlet，需要在浏览器中输入以下 URL：

```
http://localhost:8080/appl/Primitive
```

要调用 ModernServlet，需要在浏览器中输入以下 URL：

```
http://localhost:8080/appl/Modern
```

## 15.4 小结

Tomcat 用于不同的配置环境下。通过使用 Digester 对象将 XML 元素转换为 Java 对象使得用户可以通过编写 server.xml 文件来方便地配置 Tomcat。此外，web.xml 文档用于配置 servlet/JSP 应用程序。Tomcat 必须要解析 web.xml 文档，并基于 XML 文档中的元素配置 Context 对象。Digester 库很优雅地解决了这个问题。

## 第 16 章

# 关闭钩子

在很多实际应用环境中，当用户关闭了应用程序时，需要做一些善后清理工作。但问题是，用户有时并不会按照推荐的方法关闭应用程序，很有可能不做清理工作。例如，在 Tomcat 的部署应用中，通过实例化一个 Server 对象来启动 servlet 容器，调用其 start() 方法，然后逐个调用组件的 start() 方法。正常情况下，为了让 Server 对象能够关闭这些已经启动的组件，你应该像第 14 章介绍的那样，发送关闭命令。如果你只是简单地突然退出，例如在应用程序运行过程中关闭控制台，可能会发生一些意想不到的事情。

幸运的是，Java 为程序员提供了一种优雅的方法可以在关闭过程中执行一些代码，这样就能确保那些负责善后处理的代码肯定能够执行。本章将展示如何使用关闭钩子来确保清理代码总是能够执行，无论用户如何终止应用程序。

在 Java 中，虚拟机会对两类事件进行响应，然后执行关闭操作：

- 当调用 System.exit() 方法或程序的最后一个非守护进程线程退出时，应用程序正常退出；
- 用户突然强制虚拟机中断运行，例如用户按 CTRL+C 快捷键或在未关闭 Java 程序的情况下，从系统中退出。

幸运的是，虚拟机在执行关闭操作时，会经过以下两个阶段：

1) 虚拟机启动所有已经注册的关闭钩子，如果有的话。关闭钩子是先前已经通过 Runtime 类注册的线程，所有的关闭钩子会并发执行，直到完成任务；

2) 虚拟机根据情况调用所有没有被调用过的终结器 (finalizer)。

本章重点说明第一个阶段，因为该阶段允许程序员告诉虚拟机在应用程序中执行一些清理代码。关闭钩子很简单，只是 java.lang.Thread 类的一个子类的实例。创建关闭钩子很简单：

- 1) 创建 Thread 类的一个子类；
- 2) 实现你自己的 run() 方法，当应用程序（正常或突然）关闭时，会调用此方法；
- 3) 在应用程序中，实例化关闭钩子类；
- 4) 使用当前 Runtime 类的 addShutdownHook() 方法注册关闭钩子。

也许你已经注意到了，不需要像启动其他线程一样调用关闭钩子的 start 方法。虚拟机会在它运行其关闭序列时启动并执行关闭钩子。

代码清单 16-1 定义了一个简单的 ShutdownHookDemo 类和一个 Thread 类 (ShutdownHook 类) 的子类。注意，ShutdownHook 类的 run() 方法只是简单地将字符串 “Shutting down” 输出到控制台上。但是，可以插入想在应用程序关闭之前执行的任何代码。

代码清单 16-1 使用关闭钩子的一个例子

```
package ex16.pyrmont.shutdownhook;

public class ShutdownHookDemo {

    public void start() {
        System.out.println("Demo");
        ShutdownHook shutdownHook = new ShutdownHook();
        Runtime.getRuntime().addShutdownHook(shutdownHook);
    }

    public static void main(String[] args) {
        ShutdownHookDemo demo = new ShutdownHookDemo();
        demo.start();
        try {
            System.in.read();
        }
        catch (Exception e) {
        }
    }

    class ShutdownHook extends Thread {
        public void run() {
            System.out.println("Shutting down");
        }
    }
}
```

在实例化 ShutdownHookDemo 类后，main() 方法会调用 start() 方法。start() 方法会创建一个关闭钩子，并通过当前运行时注册它：

```
ShutdownHook shutdownHook = new ShutdownHook();
Runtime.getRuntime().addShutdownHook(shutdownHook);
```

然后，应用程序会等待用户的输入：

```
System.in.read();
```

当用户按 Enter 键时，应用程序会退出。但是，虚拟机机会执行关闭钩子，效果是输出字符串 “Shutting down”。

## 16.1 关闭钩子的例子

现在来看另一个例子。这是一个简单的 Swing 应用程序，其类的名字为 MySwingApp，效果如图 16-1 所示。该应用程序会在它启动时创建一个临时文件，并在关闭时删除该临时文件。

代码清单 16-2 给出了 MySwingApp 类的定义。

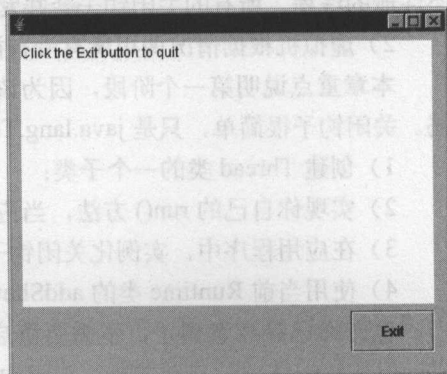


图 16-1 Swing 应用程序的小例子

代码清单 16-2 一个简单 Swing 应用程序

```
package ex16.pyrmont.shutdownhook;
import java.awt.*;
import javax.swing.*;
```



```

import java.awt.event.*;
import java.io.File;
import java.io.IOException;

public class MySwingApp extends JFrame {
    JButton exitButton = new JButton();
    JTextArea jTextArea1 = new JTextArea();
    String dir = System.getProperty("user.dir");
    String filename = "temp.txt";

    public MySwingApp() {
        exitButton.setText("Exit");
        exitButton.setBounds(new Rectangle(304, 248, 76, 37));
        exitButton.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                exitButton_actionPerformed(e);
            }
        });
        this.getContentPane().setLayout(null);
        jTextArea1.setText("Click the Exit button to quit");
        jTextArea1.setBounds(new Rectangle(9, 7, 371, 235));
        this.getContentPane().add(exitButton, null);
        this.getContentPane().add(jTextArea1, null);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.setBounds(0, 0, 400, 330);
        this.setVisible(true);
        initialize();
    }

    private void initialize() {
        // create a temp file
        File file = new File(dir, filename);
        try {
            System.out.println("Creating temporary file");
            file.createNewFile();
        } catch (IOException e) {
            System.out.println("Failed creating temporary file.");
        }
    }

    private void shutdown() {
        // delete the temp file
        File file = new File(dir, filename);
        if (file.exists()) {
            System.out.println("Deleting temporary file.");
            file.delete();
        }
    }

    void exitButton_actionPerformed(ActionEvent e) {
        shutdown();
        System.exit(0);
    }

    public static void main(String[] args) {
        MySwingApp mySwingApp = new MySwingApp();
    }
}

```

启动时，应用程序会调用其 `initialize()` 方法。然后，`initialize()` 方法会在用户目录中创建一个

个临时文件，名为“temp.txt”：

```
private void initialize() {
    // create a temp file
    File file = new File(dir, filename);
    try {
        System.out.println("Creating temporary file");
        file.createNewFile();
    }
    catch (IOException e) {
        System.out.println("Failed creating temporary file.");
    }
}
```

当用户关闭应用程序时，应用程序需要删除该临时文件。我们希望用户总是能够通过单击 Exit 按钮来退出，这样就会调用 shutdown() 方法，也就可以删除临时文件了。但是，如果用户是通过单击右上角的关闭按钮或是通过其他方法退出的，临时文件就无法删除了。

代码清单 16-3 中的类提供这个问题的解决方案。它修改了代码清单 16-2 中的代码，使用关闭钩子来删除临时文件。关闭钩子的类是一个内部类，这样它就能访问主类的所有方法。在代码清单 16-3 中，关闭钩子的 run() 方法会调用 shutdown() 方法，保证在虚拟机关闭时会调用 shutdown() 方法。

代码清单 16-3 使用关闭钩子的 Swing 应用程序

```
package ex16.pyrmont.shutdownhook;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.io.File;
import java.io.IOException;

public class MySwingAppWithShutdownHook extends JFrame {
    JButton exitButton = new JButton();
    JTextArea jTextArea1 = new JTextArea();
    String dir = System.getProperty("user.dir");
    String filename = "temp.txt";

    public MySwingAppWithShutdownHook() {
        exitButton.setText("Exit");
        exitButton.setBounds(new Rectangle(304, 248, 76, 37));
        exitButton.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                exitButton_actionPerformed(e);
            }
        });
        this.getContentPane().setLayout(null);
        jTextArea1.setText("Click the Exit button to quit");
        jTextArea1.setBounds(new Rectangle(9, 7, 371, 235));
        this.getContentPane().add(exitButton, null);
        this.getContentPane().add(jTextArea1, null);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.setBounds(0, 0, 400, 330);
        this.setVisible(true);
        initialize();
    }
}
```

```

private void initialize() {
    // add shutdown hook
    MyShutdownHook shutdownHook = new MyShutdownHook();
    Runtime.getRuntime().addShutdownHook(shutdownHook);

    // create a temp file
    File file = new File(dir, filename);
    try {
        System.out.println("Creating temporary file");
        file.createNewFile();
    }
    catch (IOException e) {
        System.out.println("Failed creating temporary file.");
    }
}

private void shutdown() {
    // delete the temp file
    File file = new File(dir, filename);
    if (file.exists()) {
        System.out.println("Deleting temporary file.");
        file.delete();
    }
}

void exitButton_actionPerformed(ActionEvent e) {
    shutdown();
    System.exit(0);
}

public static void main(String[] args) {
    MySwingAppWithShutdownHook mySwingApp = new
    MySwingAppWithShutdownHook();
}

private class MyShutdownHook extends Thread {
    public void run() {
        shutdown();
    }
}
}

```

注意代码清单 16-3 中的 `initialize()` 方法。它首先会创建内部类 `MyShutdownHook` 的一个实例，该类继承自 `java.lang.Thread` 类：

```

// add shutdown hook
MyShutdownHook shutdownHook = new MyShutdownHook();

```

一旦获得了 `MyShutdownHook` 类的实例后，就需要将其传递给 `Runtime` 类的 `addShutdownHook()` 方法，如下所示：

```

Runtime.getRuntime().addShutdownHook(shutdownHook);

```

`initialize()` 方法剩余的代码与代码清单 16-2 中的类中的 `initialize()` 方法相似。它会创建一个临时文件，并输出字符串 “Creating temporary file”：

```

// create a temp file
File file = new File(dir, filename);
try {
    System.out.println("Creating temporary file");
}

```



```
file.createNewFile();
}
catch (IOException e) {
    System.out.println("Failed creating temporary file.");
}
}
```

现在，启动代码清单 16-3 中的小应用程序。检查一下，当突然关闭应用程序时，是否总是删除临时文件。

## 16.2 Tomcat 中的关闭钩子

正如你所想的一样，Tomcat 也是通过关闭钩子来完成退出过程的。在 `org.apache.catalina.startup.Catalina` 类中，可以找到这样的代码。Catalina 类负责启动管理其他组件的 Server 对象。一个名为 `CatalinaShutdownHook` 的内部类继承自 `java.lang.Thread` 类，提供了 `run()` 方法的实现，它会调用 Server 对象的 `stop()` 方法，执行关闭操作。代码清单 16-4 给出了 `CatalinaShutdownHook` 类的定义。

代码清单 16-4 CatalinaShutdownHook 类的定义

```
protected class CatalinaShutdownHook extends Thread {
    public void run() {
        if (server != null) {
            try {
                ((Lifecycle) server).stop();
            }
            catch (LifecycleException e) {
                System.out.println("Catalina.stop: " + e);
                e.printStackTrace(System.out);
                if (e.getThrowable() != null) {
                    System.out.println("----- Root Cause -----");
                    e.getThrowable().printStackTrace(System.out);
                }
            }
        }
    }
}
```

在 Catalina 实例启动时，会实例化关闭钩子，并在一个阶段将其添加到 Runtime 类中。第 17 章将会介绍更多关于 Catalina 类的内容。

## 16.3 小结

有时候，应用程序在关闭之前应该执行一些代码清理工作。但是，你不能假设用户总是正常退出。本章介绍的关闭钩子提供了一种解决方案，确保无论用户如何关闭应用程序，清理代码总是能够得到执行。

## 第 17 章

# 启动 Tomcat

本章将重点关注启动 Tomcat 时会用到的两个类，分别是 Catalina 类和 Bootstrap 类，它们都位于 org.apache.catalina.startup 包下。Catalina 类用于启动或关闭 Server 对象，并负责解析 Tomcat 配置文件：server.xml 文件。Bootstrap 类是一个入口点，负责创建 Catalina 实例，并调用其 process() 方法。理论上，这两个类可以合并为一个。但为了支持 Tomcat 的多种运行模式，而提供了多种启动类。例如，上面说到的 Bootstrap 类是作为一个独立的应用程序运行 Tomcat 的。而另一个类 org.apache.catalina.startup.BootstrapService 可以使 Tomcat 作为一个 Windows NT 服务来运行。

为了让用户使用方便，Tomcat 附带了批处理文件和 Shell 脚本，可以方便地启动或关闭 servlet 容器。在这些批处理文件和 Shell 脚本的帮助下，用户无须为了执行 Bootstrap 类而记下 java.exe 程序的选项。相反，用户只需要运行相应的批处理文件或是 Shell 脚本即可。

17.1 节会对 Catalina 类进行介绍，然后在 17.2 节中对 Bootstrap 类进行介绍。为了能够更好地理解本章的内容，你需要了解第 14 章中关于服务器组件和服务组件的知识，第 15 章中关于 Digester 库的知识和第 16 章中关于关闭钩子的知识。本章会用两节分别介绍如何在 Windows 平台和 UNIX/Linux 平台上运行 Tomcat。一节专门用于讨论 Windows 平台上启动和关闭 Tomcat 的批处理文件，另一节解释 Unix/Linux 平台上的 Shell 脚本。

### 17.1 Catalina 类

org.apache.catalina.startup.Catalina 类是启动类。它包含一个 Digester 对象，用于解析位于 %CATALINE\_HOME%/conf 目录下的 server.xml 文件。理解了添加到 Digester 对象中的规则后，就可以自行配置 Tomcat 了。

Catalina 类还封装了一个 Server 对象，该对象有一个 Service 对象。正如第 15 章中介绍的那样，Service 对象包含有一个 Servlet 容器和一个或多个连接器。可以使用 Catalina 类来启动 / 关闭 Server 对象。

可以通过实例化 Catalina 类，并调用其 process() 方法来运行 Tomcat，但在调用该方法时，需要传入适当的参数。第 1 个参数是 start（表示要启动 Tomcat），或 stop（表示要向 Tomcat 发送一条关闭命令）。还有其他可选的参数，包括 -help、-config、-debug 和 -nonaming。

**注意** 当使用 nonaming 参数时，表示将不对 JNDI 命名提供支持。更多关于 Tomcat 中对 JNDI 命名的支持请参见 org.apache.naming 包中的信息。

一般情况下，即使 Catalina 类提供了 main() 方法作为程序的入口点，也需要使用 Bootstrap

类来实例化 Catalina 类，并调用其 process() 方法。Bootstrap 类将在 17.2 节中介绍。

代码清单 17-1 给出了 Tomcat 4 中 Catalina 类的 process() 方法的实现。

代码清单 17-1 Tomcat 4 中 Catalina 类的 process() 方法的实现

```
public void process(String args[]) {  
    setCatalinaHome();  
    setCatalinaBase();  
    try {  
        if (arguments(args))  
            execute();  
    }  
    catch (Exception e) {  
        e.printStackTrace(System.out);  
    }  
}
```

process() 方法设置了两个系统属性，分别是 catalina.home 和 catalina.base.catalina.home，默认值均为 user.dir 属性的值。catalina.base 属性的值与 catalina.home 属性的值相同。因此，它们都与 user.dir 属性的值相同。

**注意** user.dir 属性的值指明了用户的工作目录，即，会从哪个目录下调用 java 命令。更多系统属性的列表，请参见 J2SE 1.4 API 规范文档中对 java.lang.System() 类的 getProperties() 方法的介绍。

然后，process() 方法会调用 arguments() 方法，并传入参数列表。arguments() 方法处理命令行参数，如果 Catalina 对象能够继续处理的话，arguments() 方法返回 true。代码清单 17-2 给出了 arguments() 方法的实现。

代码清单 17-2 arguments() 方法的实现

```
protected boolean arguments(String args[]) {  
    boolean isConfig = false;  
    if (args.length < 1) {  
        usage();  
        return (false);  
    }  
    for (int i = 0; i < args.length; i++) {  
        if (isConfig) {  
            configFile = args[i];  
            isConfig = false;  
        }  
        else if (args[i].equals("-config")) {  
            isConfig = true;  
        }  
        else if (args[i].equals("-debug")) {  
            debug = true;  
        }  
        else if (args[i].equals("-nonaming")) {  
            useNaming = false;  
        }  
        else if (args[i].equals("-help")) {  
            usage();  
            return (false);  
        }  
    }  
}
```



```

else if (args[i].equals("start")) {
    starting = true;
}
else if (args[i].equals("stop")) {
    stopping = true;
}
else {
    usage();
    return (false);
}
return (true);
}

```

process() 方法会检查 arguments() 方法的返回值, 如果 arguments() 方法返回 true, 则调用 execute() 方法。代码清单 17-3 给出了 execute() 方法的实现。

代码清单 17-3 execute() 方法的实现

```

protected void execute() throws Exception {
    if (starting)
        start();
    else if (stopping)
        stop();
}

```

execute() 方法会调用 start() 方法来启动 Tomcat, 或调用 stop() 方法来关闭 Tomcat。这两个方法将在下面几节中讨论。

**注意** 在 Tomcat 5 中没有 execute() 方法, 会在 process() 方法中调用 start() 方法或 stop() 方法。

### 17.1.1 start() 方法

start() 方法会创建一个 Digester 实例来解析 server.xml 文件 (Tomcat 配置文件)。在解析 server.xml 文件之前, start() 方法会调用 Digester 对象的 push() 方法, 传入当前的 Catalina 对象作为参数。这样, Catalina 对象就成了 Digester 对象的内部栈中的第一个对象。解析 server.xml 文件后, 会使变量 server 引用一个 Server 对象, 默认是 org.apache.catalina.core.StandardServer 类型的对象。然后, start() 方法会调用 Server 对象的 initialize() 和 start() 方法。接着, Catalina 对象的 start() 方法会调用 Server 对象的 await() 方法, Server 对象会使用一个专用的线程来等待关闭命令。await() 方法会循环等待, 直到接收到正确的关闭命令。当 await() 方法返回时, Catalina 对象的 start() 方法会调用 Server 对象的 stop() 方法, 从而关闭 Server 对象和其他的组件。此外, start() 方法还会利用关闭钩子, 确保用户突然退出应用程序时会执行 Server 对象的 stop() 方法。

代码清单 17-4 给出了 start() 方法的实现。

代码清单 17-4 start() 方法的实现

```

protected void start() {

```

```

// Create and execute our Digester
Digester digester = createStartDigester();
File file = configFile();
try {
    InputSource is =
        new InputSource("file://" + file.getAbsolutePath());
    FileInputStream fis = new FileInputStream(file);
    is.setByteStream(fis);
    digester.push(this);
    digester.parse(is);
    fis.close();
}
catch (Exception e) {
    System.out.println("Catalina.start: " + e);
    e.printStackTrace(System.out);
    System.exit(1);
}

// Setting additional variables
if (!useNaming) {
    System.setProperty("catalina.useNaming", "false");
}
else {
    System.setProperty("catalina.useNaming", "true");
    String value = "org.apache.naming";
    String oldValue =
        System.getProperty(javax.naming.Context.URL_PKG_PREFIXES);
    if (oldValue != null) {
        value = value + ":" + oldValue;
    }
    System.setProperty(javax.naming.Context.URL_PKG_PREFIXES, value);
    value = System.getProperty
        (javax.naming.Context.INITIAL_CONTEXT_FACTORY);
    if (value == null) {
        System.setProperty
            (javax.naming.Context.INITIAL_CONTEXT_FACTORY,
             "org.apache.naming.java.javaURLContextFactory");
    }
}

// If a SecurityManager is being used, set properties for
// checkPackageAccess() and checkPackageDefinition
if( System.getSecurityManager() != null ) {
    String access = Security.getProperty("package.access");
    if( access != null && access.length() > 0 )
        access += ",";
    else
        access = "sun.";
    Security.setProperty("package.access",
        access + "org.apache.catalina.,org.apache.jasper.");
    String definition = Security.getProperty("package.definition");
    if( definition != null && definition.length() > 0 )
        definition += ",";
    else
        definition = "sun.";
    Security.setProperty("package.definition",
        definition + "org.apache.catalina.,org.apache.jasper.");
}
// FIX ME package "javax." was removed to prevent HotSpot
// fatal internal errors
definition + "java.,org.apache.catalina.,org.apache.jasper.");
}

```

```
// Replace System.out and System.err with a custom PrintStream
SystemLogHandler log = new SystemLogHandler(System.out);
System.setOut(log);
System.setErr(log);
```

```
Thread shutdownHook = new CatalinaShutdownHook();
```

```
// Start the new server
```

```
if (server instanceof Lifecycle) {
```

```
try {
```

```
    server.initialize();
```

```
    ((Lifecycle) server).start();
```

```
    try {
```

```
        // Register shutdown hook
```

```
        Runtime.getRuntime().addShutdownHook(shutdownHook);
```

```
    }
```

```
    catch (Throwable t) {
```

```
        // This will fail on JDK 1.2. Ignoring, as Tomcat can run
```

```
        // fine without the shutdown hook.
```

```
    }
```

```
    // Wait for the server to be told to shut down
```

```
    server.await();
```

```
}
```

```
catch (LifecycleException e) {
```

```
    System.out.println("Catalina.start: " + e);
```

```
    e.printStackTrace(System.out);
```

```
    if (e.getThrowable() != null) {
```

```
        System.out.println("----- Root Cause -----");
```

```
        e.getThrowable().printStackTrace(System.out);
```

```
    }
```

```
}
```

```
// Shut down the server
```

```
if (server instanceof Lifecycle) {
```

```
try {
```

```
    try {
```

```
        // Remove the ShutdownHook first so that server.stop()
```

```
        // doesn't get invoked twice
```

```
        Runtime.getRuntime().removeShutdownHook(shutdownHook);
```

```
    }
```

```
    catch (Throwable t) {
```

```
        // This will fail on JDK 1.2. Ignoring, as Tomcat can run
```

```
        // fine without the shutdown hook.
```

```
    }
```

```
    ((Lifecycle) server).stop();
```

```
}
```

```
catch (LifecycleException e) {
```

```
    System.out.println("Catalina.stop: " + e);
```

```
    e.printStackTrace(System.out);
```

```
    if (e.getThrowable() != null) {
```

```
        System.out.println("----- Root Cause -----");
```

```
        e.getThrowable().printStackTrace(System.out);
```

```
    }
```

```
}
```



### 17.1.2 stop() 方法

stop() 方法用来关闭 Catalina 和 Server 对象。代码清单 17-5 给出了 stop() 方法的实现。

代码清单 17-5 stop() 方法的实现

```
protected void stop() {
    // Create and execute our Digester
    Digester digester = createStopDigester();
    File file = configFile();
    try {
        InputSource is =
            new InputSource("file://" + file.getAbsolutePath());
        FileInputStream fis = new FileInputStream(file);
        is.setByteStream(fis);
        digester.push(this);
        digester.parse(is);
        fis.close();
    }
    catch (Exception e) {
        System.out.println("Catalina.stop: " + e);
        e.printStackTrace(System.out);
        System.exit(1);
    }

    // Stop the existing server
    try {
        Socket socket = new Socket("127.0.0.1", server.getPort());
        OutputStream stream = socket.getOutputStream();
        String shutdown = server.getShutdown();
        for (int i = 0; i < shutdown.length(); i++)
            stream.write(shutdown.charAt(i));
        stream.flush();
        stream.close();
        socket.close();
    }
    catch (IOException e) {
        System.out.println("Catalina.stop: " + e);
        e.printStackTrace(System.out);
        System.exit(1);
    }
}
```

注意，stop() 方法调用 createStopDigester() 方法来创建一个 Digester 实例，并调用该实例的 push() 方法将当前 Catalina 对象压入到 Digester 对象的内部栈中，使用 Digester 对象解析 Tomcat 的配置文件。添加到 Digester 对象中的规则将在 17.1.4 节介绍。

然后，stop() 方法会向正在运行的 Server 对象发送关闭命令，以关闭 Server 对象。

### 17.1.3 启动 Digester 对象

Catalina 类的 createStartDigester 方法创建了一个 Digester 实例，然后为其添加规则，以解析 server.xml 文件。server.xml 文件用来配置 Tomcat，位于 %CATALINE\_HOME%/conf 目录下。添加到 Digester 对象中的规则是理解 Tomcat 配置的关键。

代码清单 17-6 给出了 createStartDigester() 方法的实现。

代码清单 17-6 createStartDigester() 方法的实现

```

protected Digester createStartDigester() {
    // Initialize the digester
    Digester digester = new Digester();
    if (debug)
        digester.setDebug(999);
    digester.setValidating(false);

    // Configure the actions we will be using
    digester.addObjectCreate("Server",
        "org.apache.catalina.core.StandardServer", "className");
    digester.addSetProperties("Server");
    digester.addSetNext("Server", "setServer",
        "org.apache.catalina.Server");

    digester.addObjectCreate("Server/GlobalNamingResources",
        "org.apache.catalina.deploy.NamingResources");
    digester.addSetProperties("Server/GlobalNamingResources");
    digester.addSetNext("Server/GlobalNamingResources",
        "setGlobalNamingResources",
        "org.apache.catalina.deploy.NamingResources");

    digester.addObjectCreate("Server/Listener", null, "className");
    digester.addSetProperties("Server/Listener");
    digester.addSetNext("Server/Listener",
        "addLifecycleListener",
        "org.apache.catalina.LifecycleListener");

    digester.addObjectCreate("Server/Service",
        "org.apache.catalina.core.StandardService", "className");
    digester.addSetProperties("Server/Service");
    digester.addSetNext("Server/Service", "addService",
        "org.apache.catalina.Service");

    digester.addObjectCreate("Server/Service/Listener",
        null, "className");
    digester.addSetProperties("Server/Service/Listener");
    digester.addSetNext("Server/Service/Listener",
        "addLifecycleListener", "org.apache.catalina.LifecycleListener");

    digester.addObjectCreate("Server/Service/Connector",
        "org.apache.catalina.connector.http.HttpConnector",
        "className");
    digester.addSetProperties("Server/Service/Connector");
    digester.addSetNext("Server/Service/Connector",
        "addConnector", "org.apache.catalina.Connector");

    digester.addObjectCreate("Server/Service/Connector/Factory",
        "org.apache.catalina.net.DefaultServerSocketFactory",
        "className");
    digester.addSetProperties("Server/Service/Connector/Factory");
    digester.addSetNext("Server/Service/Connector/Factory",
        "setFactory", "org.apache.catalina.net.ServerSocketFactory");

    digester.addObjectCreate("Server/Service/Connector/Listener",
        null, "className");
    digester.addSetProperties("Server/Service/Connector/Listener");
    digester.addSetNext("Server/Service/Connector/Listener",
        "addLifecycleListener", "org.apache.catalina.LifecycleListener");
}

```

```

// Add RuleSets for nested elements
digester.addRuleSet(
    new NamingRuleSet("Server/GlobalNamingResources/"));
digester.addRuleSet(new EngineRuleSet("Server/Service/"));
digester.addRuleSet(new HostRuleSet("Server/Service/Engine/"));
digester.addRuleSet(new
    ContextRuleSet("Server/Service/Engine/Default"));
digester.addRuleSet(
    new NamingRuleSet("Server/Service/Engine/DefaultContext/"));
digester.addRuleSet(
    new ContextRuleSet("Server/Service/Engine/Host/Default"));
digester.addRuleSet(
    new NamingRuleSet("Server/Service/Engine/Host/DefaultContext/"));
digester.addRuleSet(
    new ContextRuleSet("Server/Service/Engine/Host/"));
digester.addRuleSet(
    new NamingRuleSet("Server/Service/Engine/Host/Context/"));
digester.addRule("Server/Service/Engine",
    new SetParentClassLoaderRule(digester, parentClassLoader));

return (digester);
}

```

`createStartDigester()` 方法会创建 `org.apache.commons.digester.Digester` 类的一个实例，并为其添加规则。

前 3 条规则用于解析 `server.xml` 文件的 `server` 元素。可能你已经知道了，`server` 元素是 `server.xml` 文件的根元素。下面是为 `server` 模式添加的规则。

```

digester.addObjectCreate("Server",
    "org.apache.catalina.core.StandardServer", "className");
digester.addSetProperties("Server");
digester.addSetNext("Server", "setServer",
    "org.apache.catalina.Server");

```

第 1 条规则表明，在遇到 `server` 元素时，`Digester` 对象要创建 `org.apache.catalina.core.StandardServer` 类的一个实例。例外是，如果 `server` 元素有一个名为 `className` 的属性，那么 `className` 属性的值就是必须实例化的类的名称。

第 2 条规则指明要对 `Server` 对象的指定属性名设置同名的属性值。

第 3 条规则将 `Server` 对象压入到 `Digester` 对象的内部栈中，并与栈中的下一个对象相关联。在这里，下一个对象是 `Catalina` 实例，调用其 `setServer()` 方法与 `Server` 对象相关联。那 `Catalina` 实例是如何放到 `Digester` 对象的内部栈中的呢？在 `start()` 方法的开始部分，在解析 `server.xml` 文件之前，会调用 `Digester` 对象的 `push()` 方法将 `Catalina` 对象压入栈：

```

digester.push (this);

```

现在你应该已经可以理解后面的规则的意思了。如果你还理解不了的话，请再阅读一遍第 15 章的内容。

#### 17.1.4 关闭 `Digester` 对象

`createStopDigester()` 方法返回一个 `Digester` 对象来关闭 `Server` 对象。代码清单 17-7 给出了



createStopDigester() 方法的实现。

代码清单 17-7 createStopDigester() 方法的实现

```
protected Digester createStopDigester() {
    // Initialize the digester
    Digester digester = new Digester();
    if (debug)
        digester.setDebug(999);

    // Configure the rules we need for shutting down
    digester.addObjectCreate("Server",
        "org.apache.catalina.core.StandardServer", "className");
    digester.addSetProperties("Server");
    digester.addSetNext("Server", "setServer",
        "org.apache.catalina.Server");
    return (digester);
}
```

与启动 Digester 对象不同，关闭 Digester 对象只对 XML 文档的根元素感兴趣。

## 17.2 Bootstrap 类

有一些类提供了启动 Tomcat 的入口点，其中之一是 org.apache.catalina.startup.Bootstrap 类。当运行 startup.bat 文件或 startup.sh 文件时，实际上是调用了该类的 main() 方法。main() 方法会创建 3 个类载入器，并实例化 Catalina 类。然后它调用 Catalina 实例的 process() 方法。

代码清单 17-8 给出了 Bootstrap 类的定义。

代码清单 17-8 Bootstrap 类的定义

```
package org.apache.catalina.startup;

import java.io.File;
import java.lang.reflect.Method;

/**
 * Bootstrap loader for Catalina. This application constructs a
 * class loader for use in loading the Catalina internal classes
 * (by accumulating all of the JAR files found in the "server"
 * directory under "catalina.home"), and starts the regular execution
 * of the container. The purpose of this roundabout approach is to
 * keep the Catalina internal classes (and any other classes they
 * depend on, such as an XML parser) out of the system
 * class path and therefore not visible to application level classes.
 *
 * @author Craig R. McClanahan
 * @version $Revision: 1.36 $ $Date: 2002/04/01 19:51:31 $
 */
public final class Bootstrap {
    /**
     * Debugging detail level for processing the startup.
     */
    private static int debug = 0;

    /**
     * The main program for the bootstrap.
     */
}
```

```

*
* @param args Command line arguments to be processed
*/
public static void main(String args[]) {

    // Set the debug flag appropriately
    for (int i = 0; i < args.length; i++) {
        if ("--debug".equals(args[i]))
            debug = 1;
    }

    // Configure catalina.base from catalina.home if not yet set
    if (System.getProperty("catalina.base") == null)
        System.setProperty("catalina.base", get CatalinaHome());

    // Construct the class loaders we will need
    ClassLoader commonLoader = null;
    ClassLoader catalinaLoader = null;
    ClassLoader sharedLoader = null;

    try {
        File unpacked[] = new File[1];
        File packed[] = new File[1];
        File packed2[] = new File[2];
        ClassLoaderFactory.setDebug(debug);

        unpacked[0] = new File(getCatalinaHome(),
            "common" + File.separator + "classes");
        packed2[0] = new File(getCatalinaHome(),
            "common" + File.separator + "endorsed");
        packed2[1] = new File(getCatalinaHome(),
            "common" + File.separator + "lib");
        commonLoader =
            ClassLoaderFactory.createClassLoader(unpacked, packed2, null);

        unpacked[0] = new File(getCatalinaHome(),
            "server" + File.separator + "classes");
        packed[0] = new File(getCatalinaHome(),
            "server" + File.separator + "lib");
        catalinaLoader =
            ClassLoaderFactory.createClassLoader(unpacked, packed,
            commonLoader);

        unpacked[0] = new File(getCatalinaBase(),
            "shared" + File.separator + "classes");
        packed[0] = new File(getCatalinaBase(),
            "shared" + File.separator + "lib");
        sharedLoader =
            ClassLoaderFactory.createClassLoader(unpacked, packed,
            commonLoader);
    }
    catch (Throwable t) {
        log("Class loader creation threw exception", t);
        System.exit(1);
    }

    Thread.currentThread().setContextClassLoader(catalinaLoader);

    // Load our startup class and call its process() method
    try {

```

```

SecurityClassLoader.securityClassLoader(catalinaLoader);
// Instantiate a startup class instance
if (debug >= 1)
    log("Loading startup class");
Class startupClass =
    catalinaLoader.loadClass
        ("org.apache.catalina.startup.Catalina");
Object startupInstance = startupClass.newInstance();

```

```

// Set the shared extensions class loader
if (debug >= 1)
    log("Setting startup class properties");
String methodName = "setParentClassLoader";
Class paramTypes[] = new Class[1];
paramTypes[0] = Class.forName("java.lang.ClassLoader");
Object paramValues[] = new Object[1];
paramValues[0] = sharedLoader;
Method method =
    startupInstance.getClass().getMethod(methodName, paramTypes);
method.invoke(startupInstance, paramValues);

```

```

// Call the process() method
if (debug >= 1)
    log("Calling startup class process() method");
methodName = "process";
paramTypes = new Class[1];
paramTypes[0] = args.getClass();
paramValues = new Object[1];
paramValues[0] = args;
method =
    startupInstance.getClass().getMethod(methodName, paramTypes);
method.invoke(startupInstance, paramValues);

```

```

} catch (Exception e) {
    System.out.println("Exception during startup processing");
    e.printStackTrace(System.out);
    System.exit(2);
}

```

```

/**
 * Get the value of the catalina.home environment variable.
 */

```

```

private static String getCatalinaHome() {
    return System.getProperty("catalina.home",
        System.getProperty("user.dir"));
}

```

```

/**
 * Get the value of the catalina.base environment variable.
 */
private static String getCatalinaBase() {
    return System.getProperty("catalina.base", getCatalinaHome());
}

```

```

/**
 * Log a debugging detail message.
 *
 * @param message The message to be logged
 */
private static void log(String message) {
    System.out.print("Bootstrap: ");

```



```

        System.out.println(message);
    }

    /**
     * Log a debugging detail message with an exception.
     *
     * @param message The message to be logged
     * @param exception The exception to be logged
     */
    private static void log(String message, Throwable exception) {
        log(message);
        exception.printStackTrace(System.out);
    }
}

```

Bootstrap 类有 4 个静态方法，分别是两个 log() 方法、getCatalinaHome() 方法和 getCatalinaBase() 方法。getCatalinaHome() 方法的实现如下所示：

```

return System.getProperty("catalina.home",
    System.getProperty("user.dir"));

```

其基本含义是，如果先前没有设置过 catalina.home 属性的值，它就返回 user.dir 属性的值。

getCatalinaBase() 方法的实现如下所示：

```

return System.getProperty("catalina.base", getCatalinaHome());

```

其基本含义是，如果 catalina.base 属性的值为空，就返回 catalina.home 属性的值。

从 Bootstrap 类的 main() 方法中会调用 getCatalinaHome() 方法和 getCatalinaBase() 方法。

此外，在 main() 方法中还会为不同目的而创建 3 个类载入器。使用多个类载入器的目的是为了防止应用程序中的类（包括 servlet 类和 Web 应用程序中的其他辅助类）使用 WEB-INF/classes 目录和 WEB-INF/lib 目录之外的类。部署到 %CATALINA\_HOME%/common/lib 目录下的那个 JAR 文件的类文件是可以使用的。

3 个类载入器的定义如下所示：

```

// Construct the class loaders we will need
ClassLoader commonLoader = null;
ClassLoader catalinaLoader = null;
ClassLoader sharedLoader = null;

```

对于每个类载入器都会指定一条可以访问的路径。commonLoader 类载入器可以载入 %CATALINA\_HOME%/common/classes 目录、%CATALINA\_HOME%/common/endorsed 目录和 %CATALINA\_HOME%/common/lib 目录下的 Java 类：

```

try {
    File unpacked[] = new File[1];
    File packed[] = new File[1];
    File packed2[] = new File[2];
    ClassLoaderFactory.setDebug(debug);

    unpacked[0] = new File(getCatalinaHome(),
        "common" + File.separator + "classes");
    packed[0] = new File(getCatalinaHome(),
        "common" + File.separator + "endorsed");
}

```

```
packed2[1] = new File(getCatalinaHome(),
    "common" + File.separator + "lib");
commonLoader =
    ClassLoaderFactory.createClassLoader(unpacked, packed2, null);
```

catalinaLoader 类载入器负责载入运行 Catalina servlet 容器所需要的类。它可载入 %CATALINA\_HOME%/server/classes 目录、%CATALINA\_HOME%/server/lib 目录以及 commonLoader 类载入器可以访问的所有目录中的 Java 类：

```
unpacked[0] = new File(getCatalinaHome(),
    "server" + File.separator + "classes");
packed[0] = new File(getCatalinaHome(),
    "server" + File.separator + "lib");
catalinaLoader =
    ClassLoaderFactory.createClassLoader(unpacked, packed,
    commonLoader);
```

sharedLoader 类可以载入 %CATALINA\_HOME%/shared/classes 目录和 %CATALINA\_HOME%/shared/lib 目录，以及 commonLoader 类载入器可以访问的目录下的类。在 Tomcat 中，每个 Web 应用程序中与 Context 容器相关联的每个类载入器的父类载入器都是 sharedLoader 类载入器：

```
unpacked[0] = new File(getCatalinaBase(),
    "shared" + File.separator + "classes");
packed[0] = new File(getCatalinaBase(),
    "shared" + File.separator + "lib");
sharedLoader =
    ClassLoaderFactory.createClassLoader(unpacked, packed,
    commonLoader);
}
catch (Throwable t) {
    log("Class loader creation threw exception", t);
    System.exit(1);
}
```

注意，sharedLoader 类载入器并不能访问 Catalina 的内部类，或 CLASSPATH 环境变量指定的类路径中的类。更多关于类载入器的内容请参见第 8 章。

在创建了 3 个类载入器之后，main() 方法会载入 Catalina 类并创建它的一个实例，然后再将其赋值给 startupInstance 变量：

```
Class startupClass =
    catalinaLoader.loadClass
        ("org.apache.catalina.startup.Catalina");
Object startupInstance = startupClass.newInstance();
```

然后，它调用 setParentClassLoader() 方法，并将 sharedLoader 类载入器作为参数传入：

```
// Set the shared extensions class loader
if (debug >= 1)
    log("Setting startup class properties");
String methodName = "setParentClassLoader";
Class paramTypes[] = new Class[1];
paramTypes[0] = Class.forName("java.lang.ClassLoader");
Object paramValues[] = new Object[1];
paramValues[0] = sharedLoader;
Method method =
    startupInstance.getClass().getMethod(methodName, paramTypes);
method.invoke(startupInstance, paramValues);
```

最后，main() 方法会调用 Catalina 对象的 process() 方法：

```
// Call the process() method
if (debug >= 1)
    log("Calling startup class process() method");
methodName = "process";
paramTypes = new Class[1];
paramTypes[0] = args.getClass();
paramValues = new Object[1];
paramValues[0] = args;
method =
    startupInstance.getClass().getMethod(methodName, paramTypes);
method.invoke(startupInstance, paramValues);
```

## 17.3 在 Windows 平台上运行 Tomcat

正如你在前面几节中学到的，可以调用 Bootstrap 类将 Tomcat 作为一个独立的应用程序来运行。在 Windows 平台上，可以调用 startup.bat 批处理文件来启动 Tomcat，或运行 shutdown.bat 批处理文件来关闭 Tomcat。这两个批处理文件都位于 %CATALINA\_HOME%\bin 目录下。本节将对批处理脚本进行介绍。如果你已经对 DOS 命令不太熟悉，那么你需要认真阅读 17.3.1 节。

### 17.3.1 如何编写批处理文件

本节将对批处理文件进行介绍，这样你才能理解用来启动或关闭 Tomcat 的批处理文件。特别地，本节将会对 rem、if、echo、goto、label 等命令进行介绍。当然，本节介绍的内容些并不能涵盖所有的内容，你还需要查询一些其他的资料。

首先，批处理文件的后缀名必须为“.bat”。可以从 Windows Explorer 中双击一个批处理文件，也可以在 DOS 控制台中键入批处理文件的名字来调用它。调用批处理文件后，文件中的每一行命令都会解释。下面将会对 Tomcat 的批处理文件中用到的批处理命令进行介绍。

**注意** DOS 命令及环境变量是区分大小写的。

**rem**

该命令用于注释。解释器将不会执行以 rem 命令开始的行。

**pause**

pause 命令用于暂停正在执行的批处理文件，并提示用户按某个键，然后程序会继续运行。

**echo**

该命令用于在 DOS 控制台上显示一段文本。例如，下面的命令将在控制台上输出“Hello World”，然后暂停程序。之所以暂停程序是为了能够使控制台将输出的文本显示出来。

```
echo Hello World
pause
```

如果想要显示环境变量的值，需要在环境变量的前后添加“%”。例如，下面的命令将输出变量 myVar 的值：

```
echo %myVar%.
```



如果想要输出操作系统的名字，可以使用如下的命令：

```
echo %OS%
```

```
echo off
```

使用“echo off”可以防止将批处理文件中的具体命令输出，而只会输出执行结果。但是，“echo off”命令本身还是会显示出来。如果想要将“echo off”也隐藏起来，需要使用“@echo off”命令。

```
@echo off
```

该命令与“echo off”命令类似，但它会连“echo off”命令也隐藏起来。

```
set
```

set 命令用来设置用户定义或命名的环境变量。在批处理文件中设置的环境变量是临时存储在内存中的，在批处理文件执行完成后就会销毁。

例如，下面的 set 命令创建了一个名为“THE\_KING”的环境变量，将其值设置为“Elvis”，并输出到控制台上：

```
set THE_KING=Elvis
```

```
echo %THE_KING%
```

```
pause
```

**注意** 为了获取变量的值，需要在变量的前后添加“%”符号。例如，“echo %the\_king%”可以输出变量 the\_king 的值。

```
label
```

使用冒号设置一个标签。然后可以将标签传递给 goto 命令，这样程序就会跳转到标签指定的位置。下面的语句定义了一个名为“end”的标签：

```
:end
```

有关于标签的更多例子请参见对 goto 命令的介绍。

```
goto
```

goto 命令强制批处理文件跳转到指定标签所在的位置继续执行。示例如下：

```
echo Start
```

```
goto end
```

```
echo I can guarantee this line will not be executed
```

```
:end
```

```
echo End
```

```
pause
```

在第 1 行上输出了“Start”之后，批处理文件会执行 goto 命令，跳转到 end 标签后的语句继续执行。结果是，跳过第 3 行内容，没有执行它。

```
if
```

if 用来执行条件测试。有以下 3 种用法：

1) 测试变量的值。

2) 测试文件是否存在。

### 3) 测试错误值。

使用下面的命令格式来测试一个变量的值：

```
if variable==value nextCommand
```

例如，下面的 if 语句将会测试变量 myVar 的值是不是 3。如果是，则它在控制台上输出 “Correct”：

```
set myVar=3
if %myVar%==3 echo Correct
```

运行上面的命令时，会对变量 myVar 的值进行判断，并输出 “Correct”。

可以使用下面的命令格式来测试文件是否存在：

```
if exist c:\temp\myFile.txt goto start
```

如果在 c:\temp 目录下存在 myFile.txt 文件，程序就会跳转到 start 标签所在位置继续执行。

也可以使用 not 关键字来对一条语句取反。

not

not 关键字用来对一条语句取反。例如，下面的命令在变量 myVar 的值不是 3 时会输出 “Correct”：

```
set myVar=3
if not %myVar%==3 echo Correct
pause
```

当 c:\temp 目录下不存在 myFile.txt 文件时，下面的命令会跳转到标签 end 所在的位置继续执行：

```
if not exist c:\temp\myFile.txt goto end
```

exist

当测试文件是否存在时，会使用到 if 语句和 exist 命令。示例程序请参见 if 命令的例子。

### 接收参数

可以给批处理文件传递参数，并使用 %1 引用第 1 个参数，%2 引用第 2 个参数，以此类推。

例如，下面的命令将在控制台中输出第 1 个参数的值：

```
echo %1
```

如果批处理文件的名字是 test.bat，并使用 “test Hello” 命令来调用它，那么将会在控制台中输出 “Hello”。

下面的批处理文件会对第 1 个参数的值进行检查。如果第 1 个参数是 start，它就会输出 “Starting application”；如果第 1 个参数是 “stop”，就输出 “Stopping application”；否则，输出 “Invalid parameter”：

```
echo off
if %1==start goto start

if %1==stop goto stop
goto invalid

:start
echo Starting application
goto end
```

```

:stop
echo Stopping application
goto end

:invalid
echo Invalid parameter

:end

```

若要检查运行批处理文件时，是否附带参数，可以将“%1”与空字符串进行比较。例如，对于下面的批处理文件，如果运行时没有使用参数，就会在控制台上输出“No parameter”：

```

if "%1"==" " echo No parameter
或
if ""%1""="" echo No parameter

```

#### shift

shift 命令用来将参数向后移动一位，即将 %2 的值复制给 %1，%3 的值复制给 %2，以此类推。例如，下面的代码使用一条 shift 命令：

```

echo off
shift
echo %1
echo %2

```

如果在运行批处理文件时，附加了 3 个参数 a、b 和 c，那么上面的命令会有如下输出：

```

b
c

```

移动之后，要使用 %0 来引用第 1 个参数，而现在最后一个参数则失效了。

#### call

call 命令用来调用另一条命令。

#### setLocal

在批处理文件中使用 setLocal 对环境变量的修改只在当前批处理脚本中有效。当遇到 endLocal 命令后，在批处理文件的末尾修改的环境变量的值会恢复成原来的值。

#### start

打开一个新的 Windows 控制台，并可以为这个新的控制台指定一个名字，例如：

```
start "Title"
```

此外，在“Title”后面，还有传入一条将要在新的控制台中执行的命令，如下所示：

```
start "Title" commandName
```

### 17.3.2 catalina.bat 批处理文件

catalina.bat 批处理文件用来启动或关闭 Tomcat。另外两个文件（startup.bat 和 shutdown.bat）提供了更简单地启动和关闭 Tomcat 的方法。实际上，startup.bat 和 shutdown.bat 都会调用



catalina.bat 文件，并传入相应的参数。

在 %CATALINA\_HOME%/bin 目录下，需要以下面的语法格式调用 catalina.bat 脚本：

```
catalina
```

或使用下面的语法从 %CATALINA\_HOME%/bin 目录下调用该脚本：

```
bin\catalina
```

在两种情况下，参数 command 的可选值包括：

- debug, 在调试器中启动 Catalina;
- debug -security, 在使用安全管理器的情况下调试 Catalina;
- embedded, 以嵌入模式启动 Catalina;
- jpda start, 在 JPDA 调试器下启动 Catalina;
- run, 在当前窗口中启动 Catalina;
- run -security, 在当前窗口中，通过安全管理器启动 Catalina;
- start, 在新窗口总启动 Catalina;
- start -security, 在新窗口中通过安全管理器启动 Catalina;
- stop, 关闭 Catalina。

例如，要想在新窗口中启动 Catalina，可以使用如下命令：

```
catalina start
```

代码清单 17-9 给出了 catalina.bat 批处理文件的内容。

代码清单 17-9 catalina.bat 文件

```
@echo off
if "%OS%" == "Windows_NT" setlocal
rem -----
rem Start/Stop Script for the CATALINA Server
rem
rem Environment Variable Prerequisites
rem
rem   CATALINA_HOME   May point at your Catalina "build" directory.
rem
rem   CATALINA_BASE   (Optional) Base directory for resolving dynamic
portions
rem                   of a Catalina installation. If not present,
rem resolves to      the same directory that CATALINA_HOME points to.
rem
rem   CATALINA_OPTS   (Optional) Java runtime options used when the
"start",
rem                   "stop", or "run" command is executed.
rem
rem   CATALINA_TMPDIR (Optional) Directory path location of temporary
directory
rem                   the JVM should use (java.io.tmpdir). Defaults to
rem                   %CATALINA_BASE%\temp.
rem
rem   JAVA_HOME       Must point at your Java Development Kit
installation.
```

```

rem
rem  JAVA_OPTS          (Optional) Java runtime options used when the
"start",
rem
rem                    "stop", or "run" command is executed.
rem
rem  JSSE_HOME          (Optional) May point at your Java Secure Sockets
Extension
rem                    (JSSE) installation, whose JAR files will be
added to the
rem                    system class path used to start Tomcat.
rem
rem  JPDA_TRANSPORT      (Optional) JPDA transport used when the "jpda
start"
rem                    command is executed. The default is "dt_shmem".
rem
rem  JPDA_ADDRESS        (Optional) Java runtime options used when the
"jpda start"
rem                    command is executed. The default is "jdbconn".
rem
rem $Id: catalina.bat,v 1.3 2002/08/04 18:19:43 patrickl Exp $
rem -----

rem Guess CATALINA_HOME if not defined
if not "%CATALINA_HOME%" == "" goto gotHome
set CATALINA_HOME=.
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
set CATALINA_HOME=..
:gotHome
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
echo The CATALINA_HOME environment variable is not defined correctly
echo This environment variable is needed to run this program
goto end
:okHome

rem Get standard environment variables
if exist "%CATALINA_HOME%\bin\setenv.bat" call
"%CATALINA_HOME%\bin\setenv.bat"

rem Get standard Java environment variables
if exist "%CATALINA_HOME%\bin\setclasspath.bat" goto okSetclasspath
echo Cannot find %CATALINA_HOME%\bin\setclasspath.bat
echo This file is needed to run this program
goto end
:okSetclasspath
set BASEDIR=%CATALINA_HOME%
call "%CATALINA_HOME%\bin\setclasspath.bat"

rem Add on extra jar files to CLASSPATH
if "%JSSE_HOME%" == "" goto noJsse
set
CLASSPATH=%CLASSPATH%;%JSSE_HOME%\lib\jcert.jar;%JSSE_HOME%\lib\jnet.jar;
r;%JSSE_HOME%\lib\jsse.jar
:noJsse
set CLASSPATH=%CLASSPATH%;%CATALINA_HOME%\bin\bootstrap.jar

if not "%CATALINA_BASE%" == "" goto gotBase
set CATALINA_BASE=%CATALINA_HOME%
:gotBase

if not "%CATALINA_TMPDIR%" == "" goto gotTmpdir
set CATALINA_TMPDIR=%CATALINA_BASE%\temp
:gotTmpdir

```

```

rem ----- Execute The Requested Command -----
-----
echo Using CATALINA_BASE:    %CATALINA_BASE%
echo Using CATALINA_HOME:    %CATALINA_HOME%
echo Using CATALINA_TMPDIR:  %CATALINA_TMPDIR%
echo Using JAVA_HOME:        %JAVA_HOME%

set _EXECJAVA=%_RUNJAVA%
set MAINCLASS=org.apache.catalina.startup.Bootstrap
set ACTION=start
set SECURITY_POLICY_FILE=
set DEBUG_OPTS=
set JPDA=

if not "%1" == "jpda" goto noJpda
set JPDA=jpda
if not "%JPDA_TRANSPORT%" == "" goto gotJpdaTransport
set JPDA_TRANSPORT=dt_shmem
:gotJpdaTransport
if not "%JPDA_ADDRESS%" == "" goto gotJpdaAddress
set JPDA_ADDRESS=jdbconn
:gotJpdaAddress
shift
:noJpda

if "%1" == "debug" goto doDebug
if "%1" == "embedded" goto doEmbedded
if "%1" == "run" goto doRun
if "%1" == "start" goto doStart
if "%1" == "stop" goto doStop

echo Usage: catalina ( commands ... )
echo commands:
echo debug          Start Catalina in a debugger
echo debug -security Debug Catalina with a security manager
echo embedded       Start Catalina in embedded mode
echo jpda start     Start Catalina under JPDA debugger
echo run           Start Catalina in the current window
echo run -security Start in the current window with security
manager
echo start         Start Catalina in a separate window
echo start -security Start in a separate window with security
manager
echo stop         Stop Catalina
goto end

:doDebug
shift
set _EXECJAVA=%_RUNJDB%
set DEBUG_OPTS=-sourcepath "%CATALINA_HOME%\..\..\jakarta-tomcat-4.0\catalina\src\share"
if not "%1" == "-security" goto execCmd
shift
echo Using Security Manager
set SECURITY_POLICY_FILE=%CATALINA_BASE%\conf\catalina.policy
goto execCmd

:doEmbedded
shift
set MAINCLASS=org.apache.catalina.startup.Embedded
goto execCmd

```



```

:doRun
shift
if not "%1" == "-security" goto execCmd
shift
echo Using Security Manager
set SECURITY_POLICY_FILE=%CATALINA_BASE%\conf\catalina.policy
goto execCmd

:doStart
shift
if not "%OS%" == "Windows_NT" goto noTitle
set _EXECJAVA=start "Tomcat" %_RUNJAVA%
goto gotTitle
:noTitle
set _EXECJAVA=start %_RUNJAVA%
:gotTitle
if not "%1" == "-security" goto execCmd
shift
echo Using Security Manager
set SECURITY_POLICY_FILE=%CATALINA_BASE%\conf\catalina.policy
goto execCmd

:doStop
shift
set ACTION=stop
goto execCmd

:execCmd
rem Get remaining unshifted command line arguments and save them in the
set CMD_LINE_ARGS=
:setArgs
if "%1"==" " goto doneSetArgs
set CMD_LINE_ARGS=%CMD_LINE_ARGS% %1
shift
goto setArgs
:doneSetArgs

rem Execute Java with the applicable properties
if not "%JPDA%" == "" goto doJpda
if not "%SECURITY_POLICY_FILE%" == "" goto doSecurity
%_EXECJAVA% %JAVA_OPTS% %CATALINA_OPTS% %DEBUG_OPTS% -
Djava.endorsed.dirs="%JAVA_ENDORSED_DIRS%" -classpath "%CLASSPATH%" -
Dcatalina.base="%CATALINA_BASE%" -Dcatalina.home="%CATALINA_HOME%" -
Djava.io.tmpdir="%CATALINA_TMPDIR%" %MAINCLASS% %CMD_LINE_ARGS%
%ACTION%
goto end
:doSecurity
%_EXECJAVA% %JAVA_OPTS% %CATALINA_OPTS% %DEBUG_OPTS% -
Djava.endorsed.dirs="%JAVA_ENDORSED_DIRS%" -classpath "%CLASSPATH%" -
Djava.security.manager -Djava.security.policy="%SECURITY_POLICY_FILE%" -
-Dcatalina.base="%CATALINA_BASE%" -Dcatalina.home="%CATALINA_HOME%" -
Djava.io.tmpdir="%CATALINA_TMPDIR%" %MAINCLASS% %CMD_LINE_ARGS%
%ACTION%
goto end
:doJpda
if not "%SECURITY_POLICY_FILE%" == "" goto doSecurityJpda
%_EXECJAVA% %JAVA_OPTS% %CATALINA_OPTS% -Xdebug
Xrunjwp:transport=JPDA_TRANSPORT%,address=JPDA_ADDRESS%,server=y,suspend=n %DEBUG_OPTS% -Djava.endorsed.dirs="%JAVA_ENDORSED_DIRS%" -
classpath "%CLASSPATH%" -Dcatalina.base="%CATALINA_BASE%" -
Dcatalina.home="%CATALINA_HOME%" -Djava.io.tmpdir="%CATALINA_TMPDIR%"
%MAINCLASS% %CMD_LINE_ARGS% %ACTION%

```

```

goto end
:doSecurityJpda
%_EXECJAVA% %JAVA_OPTS% %CATALINA_OPTS% -
Xrunjwp:transport=%JPDA_TRANSPORT%,address=%JPDA_ADDRESS%,server=y,sus
pend=n %DEBUG_OPTS% -Djava.endorsed.dirs="%JAVA_ENDORSED_DIRS%" -
classpath "%CLASSPATH%" -Djava.security.manager -
Djava.security.policy=="%SECURITY_POLICY_FILE%" -
Dcatalina.base="%CATALINA_BASE%" -Dcatalina.home="%CATALINA_HOME%" -
Djava.io.tmpdir="%CATALINA_TMPDIR%" %MAINCLASS% %CMD_LINE_ARGS%
%ACTION%
goto end
:end

```

catalina.bat 批处理文件首先会使用 “@echo off” 命令隐藏命令的显示。然后它检查环境变量 OS 的值是不是 “Windows\_NT”，即用户是否正在使用 Windows NT、Windows 2000，或是 Windows XP。如果是，它就调用 setLocal 命令将对环境变量的修改控制在当前批处理文件中：

```
if "%OS%" == "Windows_NT" setlocal
```

然后，如果先前没有设置变量 CATALINA\_HOME 的值，就在这里进行设置。默认情况下，变量 CATALINA\_HOME 是不存在的，因为这对于运行 Tomcat 并不是必需的。

如果先前没有设置过变量 CATALINA\_HOME，那么批处理文件会猜测是从哪个目录进行调用的。首先，它会假设 catalina.bat 文件是在安装目录中进行调用的。那么，在 bin 命令下就一定会存在一个名为 “catalina.bat” 的文件：

```

if not "%CATALINA_HOME%" == "" goto gotHome
set CATALINA_HOME=.
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome

```

如果在当前目录的子目录 bin 下没有找到 catalina.bat 文件，那么就不可能是在安装目录下调用的 catalina.bat 文件。然后，批处理文件会再猜测一次。这次，它还会假设 catalina.bat 文件是在安装目录的 bin 命令下调用的，但是会将变量 CATALINA\_HOME 设置为当前目录的父目录，并检查 catalina.bat 文件是否存在于 bin 目录中：

```

set CATALINA_HOME=..
:gotHome
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome

```

如果猜测正确，就跳转到 okHome 标签所在的位置继续执行。否则，它会输出错误消息，告诉用户变量 CATALINA\_HOME 设置错误，并跳转到 end 标签所在的位置。end 标签位于批出文件的末尾，会退出脚本执行。

```

echo The CATALINA_HOME environment variable is not defined correctly
echo This environment variable is needed to run this program
goto end

```

如果 CATALINA\_HOME 设置正确，并且 setenv.bat 文件存在的话，会调用 setenv.bat 批处理脚本来设置需要的环境变量。如果 setenv.bat 文件不存在，就输出错误消息：

```

:okHome
rem Get standard environment variables
if exist "%CATALINA_HOME%\bin\setenv.bat" call
"%CATALINA_HOME%\bin\setenv.bat"

```

接下来，它检查 `setclasspath.bat` 文件是否存在。如果不存在，它输出一条错误消息，并跳转到 `end` 标签所在的位置，退出批处理文件：

```
if exist "%CATALINA_HOME%\bin\setclasspath.bat" goto okSetclasspath
echo Cannot find %CATALINA_HOME%\bin\setclasspath.bat
echo This file is needed to run this program
goto end
```

如果 `setclasspath.bat` 文件存在，它就定义变量 `BASEDIR`，并赋值为环境变量 `CATALINA_HOME` 的值。然后，调用 `setclasspath.bat` 批处理文件设置类路径：

```
:okSetclasspath
set BASEDIR=%CATALINA_HOME%
call "%CATALINA_HOME%\bin\setclasspath.bat"
```

`setclasspath.bat` 文件会检查环境变量 `JAVA_HOME` 是否设置正确，并设置将会在 `catalina.bat` 中用到的下列变量：

```
set JAVA_ENDORSED_DIRS=%BASEDIR%\common\endorsed
set CLASSPATH=%JAVA_HOME%\lib\tools.jar
set _RUNJAVA="%JAVA_HOME%\bin\java"
set _RUNJAVAW="%JAVA_HOME%\bin\javaw"
set _RUNJDB="%JAVA_HOME%\bin\jdb"
set _RUNJAVAC="%JAVA_HOME%\bin\javac"
```

然后，`catalina.bat` 文件会检查是否安装了 Java Secure Socket Extension (JSSE)，以及环境变量 `JSSE_home` 是否设置正确。如果存在环境变量 `JSSE_HOME`，就将其添加到环境变量 `CLASSPATH` 中：

```
if "%JSSE_HOME%" == "" goto noJsse
set
CLASSPATH=%CLASSPATH%;%JSSE_HOME%\lib\jcert.jar;%JSSE_HOME%\lib\jnet.jar;%JSSE_HOME%\lib\jsse.jar
```

如果先前没有设置环境变量 `JSSE_HOME`，批处理文件会继续执行下面的命令，将 `bin` 目录下的 `bootstrap.jar` 添加到环境变量 `CLASSPATH` 中：

```
:noJsse
set CLASSPATH=%CLASSPATH%;%CATALINA_HOME%\bin\bootstrap.jar
```

接着，`catalian.bat` 文件会检查变量 `CATALINA_BASE` 的值。如果先前没有设置变量 `CATALINA_BASE`，就创建 `CATALINA_BASE` 变量，并将变量 `CATALINA_HOME` 的值赋给它：

```
if not "%CATALINA_BASE%" == "" goto gotBase
set CATALINA_BASE=%CATALINA_HOME%
:gotBase
```

然后，它会检查变量 `CATALINA_TMPDIR` 的值，该变量表示变量 `CATALINA_BASE` 所表示的目录下的临时目录：

```
if not "%CATALINA_TMPDIR%" == "" goto gotTmpdir
set CATALINA_TMPDIR=%CATALINA_BASE%\temp
:gotTmpdir
```

接着，它显示一些变量的值：

```
echo Using CATALINA_BASE: %CATALINA_BASE%
echo Using CATALINA_HOME: %CATALINA_HOME%
```



```
echo Using CATALINA_TMPDIR: %CATALINA_TMPDIR%
echo Using JAVA_HOME: %JAVA_HOME%
```

然后，它将在 setclasspath.bin 中设置的变量 \_RUNJAVA 的值赋值给 \_EXECJAVA。变量 \_RUNJAVA 的值为 "%JAVA\_HOME%\bin\java"。换句话说，它就是 JAVA\_HOME 目录中 bin 目录中的 java.exe 程序：

```
set _EXECJAVA=%_RUNJAVA%
```

然后，它再设置如下的环境变量：

```
set MAINCLASS=org.apache.catalina.startup.Bootstrap
set ACTION=start
set SECURITY_POLICY_FILE=
set DEBUG_OPTS=
set JPDA=
```

接着，catalina.bat 文件会检查传入给的第 1 个参数是不是 jpda（为 Java Platform Debugger Architecture（Java 平台调试器架构）而使用）。如果是，它就设置变量 JPDA 的值为“jpda”，然后检查变量 JPDA\_TRANSPORT 和变量 JPDA\_ADDRESS 的值，并移动参数：

```
if not "%1" == "jpda" goto noJpda
set JPDA=jpda
if not "%JPDA_TRANSPORT%" == "" goto gotJpdaTransport
set JPDA_TRANSPORT=dt_shmem
:gotJpdaTransport
if not "%JPDA_ADDRESS%" == "" goto gotJpdaAddress
set JPDA_ADDRESS=jdbconn
:gotJpdaAddress
shift
```

大多数情况下，不需要使用 JPDA，因此，第 1 个参数的值必须是以下几个值之一：debug、embedded、run、start 或 stop：

```
:noJpda
if "%1" == "debug" goto doDebug
if "%1" == "embedded" goto doEmbedded
if "%1" == "run" goto doRun
if "%1" == "start" goto doStart
if "%1" == "stop" goto doStop
```

如果第 1 个参数不正确，或没有使用参数，批处理文件会输出使用说明并退出：

```
echo Usage: catalina ( commands ... )
echo commands:
echo debug          Start Catalina in a debugger
echo debug -security Debug Catalina with a security manager
echo embedded       Start Catalina in embedded mode
echo jpda start      Start Catalina under JPDA debugger
echo run            Start Catalina in the current window
echo run -security   Start in the current window with security
manager
echo start          Start Catalina in a separate window
echo start -security Start in a separate window with security
manager
echo stop           Stop Catalina
goto end
```

如果第 1 个参数是 start，就跳转到 doStart 标签；如果它是 stop，就跳转到 doStop 标签，以此类推。

跳转到 doStart 标签后, catalina.bat 文件会调用 shift 命令来检查下一个参数。如果有第 2 个参数的话, 那么它必须是 “-security”。否则, 会忽略掉它。如果下一个参数是 “-security”, 会再次调用 shift 命令, 变量 SECURITY\_POLICY\_FILE 的值会被设置为 “%CATALINA\_BASE%\conf\catalina.policy”:

```

:doStart
shift
if not "%OS%" == "Windows_NT" goto noTitle
set _EXECJAVA=start "Tomcat" %_RUNJAVA%
goto gotTitle
:noTitle
set _EXECJAVA=start %_RUNJAVA%
:gotTitle
if not "%1" == "-security" goto execCmd
shift
echo Using Security Manager
set SECURITY_POLICY_FILE=%CATALINA_BASE%\conf\catalina.policy

```

这时, 变量 \_EXECJAVA 的值可以是下面两个之一:

```

start "Tomcat" "%JAVA_HOME%\bin\java"
start "%JAVA_HOME%\bin\java"

```

然后, 它跳转到 execCmd 标签处:

```

goto execCmd

```

execCmd 标签下的命令接收剩余未移动的命令行参数, 将它们保存到变量 CMD\_LINE\_ARGS 中, 再跳转到 doneSetArgs 标签处:

```

:execCmd
set CMD_LINE_ARGS=
:setArgs
if "%1"==" " goto doneSetArgs
set CMD_LINE_ARGS=%CMD_LINE_ARGS% %1
shift
goto setArgs

```

下面是 doneSetArgs 标签处的代码:

```

:doneSetArgs
rem Execute Java with the applicable properties
if not "%JPDA%" == "" goto doJpda
if not "%SECURITY_POLICY_FILE%" == "" goto doSecurity
%_EXECJAVA% %JAVA_OPTS% %CATALINA_OPTS% %DEBUG_OPTS% -
Djava.endorsed.dirs="%JAVA_ENDORSED_DIRS%" -classpath "%CLASSPATH%" -
Dcatalina.base="%CATALINA_BASE%" -Dcatalina.home="%CATALINA_HOME%" -
Djava.io.tmpdir="%CATALINA_TMPDIR%" %MAINCLASS% %CMD_LINE_ARGS%
%ACTION%

```

例如, 在我的电脑上, 调用 catalina start, 上面的代码 (从 %\_EXECJAVA% 到 %ACTION%) 会被转换成下面的实际代码:

```

start "Tomcat" "C:\j2sdk1.4.2_02\bin\java" -
Djava.endorsed.dirs="..\common\endorsed" -classpath
"C:\j2sdk1.4.2_02\lib\tools.jar;..\bin\bootstrap.jar" -
Dcatalina.base=".." -Dcatalina.home=".." -Djava.io.tmpdir="..\temp"
org.apache.catalina.startup.Bootstrap start

```

现在,你应该已经明白在调用 catalina.bat 文件时会传入什么参数。

### 17.3.3 在 Windows 平台上启动 Tomcat

代码清单 17-10 给出了 startup.bat 批处理文件的内容,可以方便地调用 catalina.bat 文件。在调用 catalina.bat 文件时,会传入参数 start。

代码清单 17-10 startup.bat 批处理文件

```
@echo off
if "%OS%" == "Windows_NT" setlocal
rem -----
rem Start script for the CATALINA Server
rem
rem $Id: startup.bat,v 1.4 2002/08/04 18:19:43 patrickl Exp $
rem -----

rem Guess CATALINA_HOME if not defined
if not "%CATALINA_HOME%" == "" goto gotHome
set CATALINA_HOME=.
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
set CATALINA_HOME=..
:gotHome
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
echo The CATALINA_HOME environment variable is not defined correctly
echo This environment variable is needed to run this program
goto end
:okHome

set EXECUTABLE=%CATALINA_HOME%\bin\catalina.bat

rem Check that target executable exists
if exist "%EXECUTABLE%" goto okExec
echo Cannot find %EXECUTABLE%
echo This file is needed to run this program
goto end
:okExec

rem Get remaining unshifted command line arguments and save them in the
set CMD_LINE_ARGS=
:setArgs
if "%1"=="%" goto doneSetArgs
set CMD_LINE_ARGS=%CMD_LINE_ARGS% %1
shift
goto setArgs
:doneSetArgs

call "%EXECUTABLE%" start %CMD_LINE_ARGS%

:end
Stripping all rem and echo commands, you get the following:
if "%OS%" == "Windows_NT" setlocal
if not "%CATALINA_HOME%" == "" goto gotHome
set CATALINA_HOME=.
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
set CATALINA_HOME=..
:gotHome
```



```

if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
goto end
:okHome
set EXECUTABLE=%CATALINA_HOME%\bin\catalina.bat
if exist "%EXECUTABLE%" goto okExec
goto end
:okExec

rem Get remaining unshifted command line arguments and save them in the
set CMD_LINE_ARGS=
:setArgs
if "%1"=="%" goto doneSetArgs
set CMD_LINE_ARGS=%CMD_LINE_ARGS% %1
shift
goto setArgs
:doneSetArgs

call "%EXECUTABLE%" start %CMD_LINE_ARGS%

:end

```

### 17.3.4 在 Windows 平台上关闭 Tomcat

shutdown.bat 批处理文件可以方便地关闭 Tomcat。shutdown.bat 脚本文件在调用 catalina.bat 脚本时传入参数 stop。代码清单 17-11 给出了 shutdown.bat 文件的内容。

代码清单 17-11 shutdown.bat 批处理文件

```

@echo off
if "%OS%" == "Windows_NT" setlocal
rem -----
rem Stop script for the CATALINA Server
rem
rem $Id: shutdown.bat,v 1.3 2002/08/04 18:19:43 patrickl Exp $
rem -----

rem Guess CATALINA_HOME if not defined
if not "%CATALINA_HOME%" == "" goto gotHome
set CATALINA_HOME=.
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
set CATALINA_HOME=..
:gotHome
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
echo The CATALINA_HOME environment variable is not defined correctly
echo This environment variable is needed to run this program
goto end
:okHome

set EXECUTABLE=%CATALINA_HOME%\bin\catalina.bat

rem Check that target executable exists
if exist "%EXECUTABLE%" goto okExec
echo Cannot find %EXECUTABLE%
echo This file is needed to run this program
goto end
:okExec

```

```

rem Get remaining unshifted command line arguments and save them in the
set CMD_LINE_ARGS=
:setArgs
if ""$1""="" goto doneSetArgs
set CMD_LINE_ARGS=%CMD_LINE_ARGS% $1
shift
goto setArgs
:doneSetArgs

call "%EXECUTABLE%" stop %CMD_LINE_ARGS%
:end

```

## 17.4 在 Linux 平台上运行 Tomcat

在 Linux 平台上, Tomcat 使用 Shell 脚本来启动和关闭。Shell 脚本的后缀名可以是“.sh”, 位于 \$CATALINA\_HOME 的 bin 目录下。这一节将介绍其中的 4 个脚本: catalina.sh、startup.sh、shutdown.sh 和 setclasspath.sh。

本节首先对 Shell 脚本中常用的命令进行介绍。如果你对 Shell 脚本还不熟悉, 应该了解一下。然后对 catalina.sh 脚本、startup.sh 脚本、shutdown.sh 脚本和 setclasspath.sh 脚本进行介绍。setclasspath.sh 脚本由 catalina.sh 脚本来调用, 因此会在 17.4.2 节简单介绍 setclasspath.sh 脚本。

### 17.4.1 如何编写 UNIX/Linux Shell 脚本

本节将会对 Shell 脚本进行介绍, 使你能够理解 Tomcat 中的 Shell 脚本, 尤其是 catalina.sh、startup.sh、shutdown.sh 和 setclasspath.sh 这 4 个文件。关于 Shell 脚本的更多知识请查看相关资源。

实际上, Shell 脚本是一个文本文件。可以使用 vi 或其他文本编辑器来编写 Shell 脚本。但需要使用下面的语法来修改文件的访问权限, 使其可以运行:

```

$ chmod +x scriptName
$ chmod 755 scriptName

```

上面的命令会给文件的所有者赋予读、写和执行的权限, 给同一用户组和其他用户赋予读和执行的权限。

可以使用下面的命令之一来执行 Shell 脚本:

```

bash scriptName
sh scriptName
./scriptName

```

Shell 脚本是由解释器逐行执行的。Shell 脚本的扩展名可有可无, 但最常用的的扩展名是“.sh”。

下面是一些常用的命令, 学习之后可以帮助你理解 Tomcat 中的 Shell 脚本。

#### 注释

Shell 脚本使用“#”符号来表示注释内容, 注释内容不会被解释。一行开头处使用“#”符号表示这一行都是注释内容。例如:

```
# This is a comment
```

“#”符号也可以出现在一行的中间。这种情况下，“#”符号后面的内容作为注释内容。例如：

```
echo Hello # print Hello
```

## clear

使用 clear 命令来清空屏幕内容。例如，下面的脚本会先清空屏幕，再输出一个字符串：

```
clear
echo Shell scripts are useful
```

## exit

exit 命令可以用来退出当前 Shell 脚本的执行。一般情况下，exit 后面都会附加一个退出状态，其中 0 表示 Shell 脚本正常执行完成，非 0 值表示 Shell 异常退出，可能是由于某种错误引起。因此，如果因为程序运行中的某种异常而想退出，可以使用如下命令：

```
exit 1
```

## echo

使用 echo 命令可以向屏幕上输出一个字符串。例如，下面的命令会输出“Hello World”：

```
echo Hello World
```

## 调用函数

可以使用句点（.）来调用一个函数，或执行另一个 Shell 脚本。例如，下面的命令会调用当前目录下的 test.sh 脚本：

```
./test.sh
```

## 系统变量与用户自定义变量

变量名必须以字母、数字或下划线开头。使用等号就可以定义一个变量。例如，下面的命令定义了一个名为 myVar 的变量，其值为 Tootsie：

```
myVar=Tootsie
```

注意，等号的前后一定不能有空格。此外，Shell 脚本中对变量名是区分大小写的。

若想定义一个值为 NULL 的变量，可使用空字符串为其赋值，或者在等号右面留空即可：

```
myVar=
myVar=""
```

要想获取变量的值，需要在变量名的前面使用美元符号（\$）。例如，下面的命令会在屏幕上输出变量 myVar 的值：

```
echo $myVar
```

UNIX/Linux 操作系统提供了一些系统变量供用户使用。例如，变量 HOME 保存了当前用户的主目录，变量 PWD 保存了用户当前所在的目录，变量 PATH 保存了将会在那些地方查找要执行的命令等。



**警告** 在你清楚地了解了系统变量所表示的意思之前，一定不要轻易修改系统变量的值。

### expr

可以使用 `expr` 命令来计算一个表达式的值。表达式必须以反引号引用。在键盘上，反引号通常位于键位 1 的左边。下面的 Shell 脚本会计算一个数学表达式：

```
sum=`expr 100 + 200`
echo $sum
```

上面的脚本定义了一个名为 `sum` 的变量，并赋值为 300（实际上是数字 100 与 200 的和）。运行这个脚本会在控制台上输出 300。

下面是另一个脚本示例：

```
echo `expr 200 + 300`
```

这条命令会在屏幕上输出如下结果：

```
500
```

特殊表达式 ``uname`` 会在得到操作系统的名字。例如，如果你正在使用 Linux 操作系统，那么运行下面的命令会在控制台上输出 “Linux”：

```
echo `uname`
```

特殊表达式 ``dirname filePath`` 会返回指定文件所在的目录。例如，命令 ``dirname /home/user1/test.sh`` 会返回 `“/home/user1”`

### 访问参数

就像向函数中传入参数一样，也可以向 Shell 脚本中传入参数，并使用 `$1` 来引用第 1 个参数，`$2` 引用第 2 个参数，以此类推。`$#` 会返回参数的个数。`$@` 会返回所有的参数。

### shift

这条命令会将参数向后移动一位，`$1` 的值改为 `$2`，`$2` 的值改为 `$3`，以此类推。

### if ... then ... [else ...] fi

`if` 语句用来测试条件，并执行相应的命令列表。它的语法格式如下所示：

```
if condition then
    list of commands
[else
    list of commands
]
fi
```

**注意** 也可以使用 “`elif`” 来代替 “`else if`”。

例如，如果传入的第 1 个参数是 “start”，运行下面的脚本，会输出 “Starting the application”；如果传入的第 1 个参数是 “stop”，会输出 “Stopping the application”：

```
if [ "$1" = "start" ]; then
    echo Starting the application
fi
if [ "$1" = "stop" ]; then
```

```
echo Stopping the application
fi
```

**注意** 判断条件在 “[” 后面必须有一个空格，在 “]” 前面也必须有一个空格。

将 \$1 用双引号引起来可以防止解释器在发现调用脚本时没有使用参数而抛出异常。

\$0 表示的是用来调用该脚本的命令。例如，如果使用下面的命令来调用一个名为 test.sh 的脚本：

```
./test.sh
```

那么，\$0 会返回 “./test.sh”。

下面是一些用来进行条件判断的测试选项：

- -f file, 当存在文件 file 时，为 true;
- -r file, 当文件 file 可读时，为 true;
- -z string, 如果 string 是空字符串，为 true;
- -n string, 如果字符串 string 不为空，为 true;
- string1 = string2, 如果 string1 等于 string2, 为 true;
- string1 != string2, 如果 string1 不等于 string2, 为 true。

### for 循环

使用 for 循环来重复执行一些命令。语法格式如下所示：

```
for { var } in {list}
do
    list of commands
done
```

例如，下面的代码：

```
for i in 1 2 3
do
    echo iteration $i
done
```

会输出：

```
iteration 1
iteration 2
iteration 3
```

### while 循环

while 循环的语法格式如下所示：

```
while [ condition ]
do
    list of commands
done
```

例如，下面的代码：

```
n=1
while [ $n -lt 3 ];
do
```

```

    echo iteration $n
    n=$((n + 1))
done

```

会输出：

```

iteration 1
iteration 2

```

其中，`[ $n -lt 3 ]`；中 `-lt` 的意思是小于等于，因此该条件判断变量 `n` 的值是否小于等于 3。

### case

case 结构表示可以在程序中执行一些选择过程。其语法格式如下所示：

```

case $variable-name in
    pattern1)
        list of commands
        ;;
    pattern2)
        list of commands
        ;;
*)
    list of commands
    ;;
esac

```

其中，“`;;`”表示命令列表执行完成，如果上面的模式都不匹配时，“`*)`”表示所要执行的命令。

例如，下面的代码会获取操作系统的名字，并将其输出到控制台上。如果使用的不是 `cygwin`、`OS400` 或 `Linux`，将会输出 “Operating system not recognized”：

```

case "`uname`" in
    CYGWIN*) echo cygwin;;
    OS400*) echo OS400;;
    Linux*) echo Linux;;
    *) echo Operating system not recognized
esac

```

### 输出重定向

使用 “`>`” 可以将输出重定向到一个文件中。例如，下面的命令：

```
echo Hello > myFile.txt
```

会创建一个名为 `myFile.txt` 的文件，并将 “Hello” 写入到其中，不会向屏幕上输出文字。

此外，请注意，“`1>&2`” 会把标准输出上的错误消息重定向到标准错误中，“`2>&1`” 会将标准错误中的信息重定向到标准输出中。

### 条件执行

可以使用命令或条件来决定是否执行另一条命令。在这种情况下，可以使用 `&&` 和 `||`，如下所示：

```
command1 && command2
```

如果 `command1` 返回的退出状态为 0，那么会执行 `command2`。`command1` 也可以用条件判断来代替。如果条件为 `true`，那么会执行 `command2`；否则，就不执行 `command2`。

```
command1 || command2
```



如果 command1 返回了非 0 的退出状态, 会执行 command2。

```
command1 && command2 || command3
```

如果 command1 返回的退出状态为 0, 执行 command2。否则, 执行 command3。

## 17.4.2 catalina.sh 脚本

catalina.sh 文件用来在 UNIX/Linux 平台上启动或关闭 Tomcat。若要启动 Tomcat, 需要将 start 作为第 1 个参数传递给 catalina.sh 脚本。若要关闭 Tomcat, 需要将 stop 作为第 1 个参数传递给 catalina.sh 脚本。下面是有效的参数列表:

- debug, 在调试器中启动 Catalina (注意, 在 OS400 系统不可用);
- debug -security, 通过安全管理器调试 Catalina (在 OS400 系统上不可用);
- embedded, 以嵌入模式启动 Catalina;
- jpda start, 在 JPDA 调试器下启动 Catalina;
- run, 在当前窗口中启动 Catalina;
- run -security, 在安全管理器下, 使用当前窗口启动 Catalina;
- start, 使用一个新窗口启动 Catalina;
- start -security, 在安全管理器下, 使用一个新窗口启动 Catalina;
- stop, 关闭 Catalina。

代码清单 17-12 给出了 catalina.sh 文件的内容。看了前面对 Shell 命令的简单介绍, 你应该可以理解 catalina.sh 文件中的内容。

代码清单 17-12 catalina.sh 文件

```
#!/bin/sh
# -----
# Start/Stop Script for the CATALINA Server
#
# Environment Variable Prerequisites
#
#   CATALINA_HOME   May point at your Catalina "build" directory.
#
#   CATALINA_BASE   (Optional) Base directory for resolving dynamic
# portions
#                   of a Catalina installation. If not present,
# resolves to
#                   the same directory that CATALINA_HOME points to.
#
#   CATALINA_OPTS   (Optional) Java runtime options used when the
# "start",
#                   "stop", or "run" command is executed.
#
#   CATALINA_TMPDIR (Optional) Directory path location of temporary
# directory
#                   the JVM should use (java.io.tmpdir). Defaults to
#                   $CATALINA_BASE/temp.
#
#   JAVA_HOME       Must point at your Java Development Kit
# installation.
#
```

```

# JAVA_OPTS (Optional) Java runtime options used when the
"start",
# "stop", or "run" command is executed.
#
# JPDA_TRANSPORT (Optional) JPDA transport used when the "jpda
start"
# command is executed. The default is "dt_socket".
#
# JPDA_ADDRESS (Optional) Java runtime options used when the "jpda
start"
# command is executed. The default is 8000.
#
# JSSE_HOME (Optional) May point at your Java Secure Sockets
Extension
# (JSSE) installation, whose JAR files will be added
to the
# system class path used to start Tomcat.
#
# CATALINA_PID (Optional) Path of the file which should contains
the pid
# of catalina startup java process, when start (fork)
is used
#
# $Id: catalina.sh,v 1.8 2003/09/02 12:23:13 remm Exp $
# -----

# OS specific support. $var _must_ be set to either true or false.
cygwin=false
os400=false
case `uname` in
    CYGWIN*) cygwin=true;;
    OS400*) os400=true;;
    esac

# resolve links - $0 may be a softlink
PRG="$0"

while [ -h "$PRG" ]; do
    ls=`ls -ld "$PRG"`
    link=`expr "$ls" : '.*-> \(.*)$'`
    if expr "$link" : '.*/*' > /dev/null; then
        PRG="$link"
    else
        PRG=`dirname "$PRG"`/"$link"
    fi
done

# Get standard environment variables
PRGDIR=`dirname "$PRG"`
CATALINA_HOME=`cd "$PRGDIR/.." ; pwd`
if [ -r "$CATALINA_HOME"/bin/setenv.sh ]; then
    . "$CATALINA_HOME"/bin/setenv.sh
fi

# For Cygwin, ensure paths are in UNIX format before anything is
touched
if $cygwin; then
    [ -n "$JAVA_HOME" ] && JAVA_HOME=`cygpath --unix "$JAVA_HOME"`
    [ -n "$CATALINA_HOME" ] && CATALINA_HOME=`cygpath --unix
"$CATALINA_HOME"`
    [ -n "$CATALINA_BASE" ] && CATALINA_BASE=`cygpath --unix
"$CATALINA_BASE"`

```

```

[ -n "$CLASSPATH" ] && CLASSPATH=`cygpath --path --unix "$CLASSPATH"`
[ -n "$JSSE_HOME" ] && JSSE_HOME=`cygpath --path --unix "$JSSE_HOME"`
fi

# For OS400
if $os400; then
    # Set job priority to standard for interactive (interactive - 6) by
    using
    # the interactive priority - 6, the helper threads that respond to
    requests
    # will be running at the same priority as interactive jobs.
    COMMAND='chgjob job('$JOBNAME') runpty(6)'
    system $COMMAND

    # Enable multi threading
    export QIBM_MULTI_THREADED=Y
fi

# Get standard Java environment variables
if [ -r "$CATALINA_HOME"/bin/setclasspath.sh ]; then
    BASEDIR="$CATALINA_HOME"
    . "$CATALINA_HOME"/bin/setclasspath.sh
else
    echo "Cannot find $CATALINA_HOME/bin/setclasspath.sh"
    echo "This file is needed to run this program"
    exit 1
fi

# Add on extra jar files to CLASSPATH
if [ -n "$JSSE_HOME" ]; then
    CLASSPATH="$CLASSPATH": "$JSSE_HOME"/lib/jcert.jar: "$JSSE_HOME"/lib/jnet
    .jar: "$JSSE_HOME"/lib/jsse.jar
fi
CLASSPATH="$CLASSPATH": "$CATALINA_HOME"/bin/bootstrap.jar

if [ -z "$CATALINA_BASE" ]; then
    CATALINA_BASE="$CATALINA_HOME"
fi

if [ -z "$CATALINA_TMPDIR" ]; then
    # Define the java.io.tmpdir to use for Catalina
    CATALINA_TMPDIR="$CATALINA_BASE"/temp
fi

# For Cygwin, switch paths to Windows format before running java
if $cygwin; then
    JAVA_HOME=`cygpath --path --windows "$JAVA_HOME"`
    CATALINA_HOME=`cygpath --path --windows "$CATALINA_HOME"`
    CATALINA_BASE=`cygpath --path --windows "$CATALINA_BASE"`
    CATALINA_TMPDIR=`cygpath --path --windows "$CATALINA_TMPDIR"`
    CLASSPATH=`cygpath --path --windows "$CLASSPATH"`
    JSSE_HOME=`cygpath --path --windows "$JSSE_HOME"`
fi

# ----- Execute The Requested Command -----
echo "Using CATALINA_BASE:   $CATALINA_BASE"
echo "Using CATALINA_HOME:   $CATALINA_HOME"
echo "Using CATALINA_TMPDIR:  $CATALINA_TMPDIR"
echo "Using JAVA_HOME:       $JAVA_HOME"

```



```

if [ "$1" = "jpda" ] ; then
    if [ -z "$JPDA_TRANSPORT" ]; then
        JPDA_TRANSPORT="dt_socket"
    fi
    if [ -z "$JPDA_ADDRESS" ]; then
        JPDA_ADDRESS="8000"
    fi
    if [ -z "$JPDA_OPTS" ]; then
        JPDA_OPTS="-Xdebug -
Xrunjdpw:transport=$JPDA_TRANSPORT,address=$JPDA_ADDRESS,server=y,suspe
nd=n"
    fi
    CATALINA_OPTS="$CATALINA_OPTS $JPDA_OPTS"
    shift
fi

if [ "$1" = "debug" ] ; then

    if $os400; then
        echo "Debug command not available on OS400"
        exit 1
    else
        shift
        if [ "$1" = "-security" ] ; then
            echo "Using Security Manager"
            shift
            exec "$_RUNJDB" $JAVA_OPTS $CATALINA_OPTS \
                -Djava.endorsed.dirs="$JAVA_ENDORSED_DIRS" -classpath
"$CLASSPATH" \
                -sourcepath "$CATALINA_HOME"/../../jakarta-tomcat-
4.0.0/catalina/src/share \
                -Djava.security.manager \
                -Djava.security.policy=="$CATALINA_BASE"/conf/catalina.policy \
                -Dcatalina.base="$CATALINA_BASE" \
                -Dcatalina.home="$CATALINA_HOME" \
                -Djava.io.tmpdir="$CATALINA_TMPDIR" \
                org.apache.catalina.startup.Bootstrap "$@" start
        else
            exec "$_RUNJDB" $JAVA_OPTS $CATALINA_OPTS \
                -Djava.endorsed.dirs="$JAVA_ENDORSED_DIRS" -classpath
"$CLASSPATH" \
                -sourcepath "$CATALINA_HOME"/../../jakarta-tomcat-
4.0.0/catalina/src/share \
                -Dcatalina.base="$CATALINA_BASE" \
                -Dcatalina.home="$CATALINA_HOME" \
                -Djava.io.tmpdir="$CATALINA_TMPDIR" \
                org.apache.catalina.startup.Bootstrap "$@" start
        fi
    fi
elif [ "$1" = "embedded" ] ; then
    shift
    echo "Embedded Classpath: $CLASSPATH"
    exec "$_RUNJAVA" $JAVA_OPTS $CATALINA_OPTS \
        -Djava.endorsed.dirs="$JAVA_ENDORSED_DIRS" -classpath "$CLASSPATH"
\
        -Dcatalina.base="$CATALINA_BASE" \
        -Dcatalina.home="$CATALINA_HOME" \
        -Djava.io.tmpdir="$CATALINA_TMPDIR" \
        org.apache.catalina.startup.Embedded "$@"

```

```

elif [ "$1" = "run" ]; then
    shift
    if [ "$1" = "-security" ]; then
        echo "Using Security Manager"
        shift
        exec "$_RUNJAVA" $JAVA_OPTS $CATALINA_OPTS \
            -Djava.endorsed.dirs="$JAVA_ENDORSED_DIRS" -classpath
"$CLASSPATH" \
            -Djava.security.manager \
            -Djava.security.policy=="$CATALINA_BASE"/conf/catalina.policy \
            -Dcatalina.base="$CATALINA_BASE" \
            -Dcatalina.home="$CATALINA_HOME" \
            -Djava.io.tmpdir="$CATALINA_TMPDIR" \
            org.apache.catalina.startup.Bootstrap "$@" start
    else
        exec "$_RUNJAVA" $JAVA_OPTS $CATALINA_OPTS \
            -Djava.endorsed.dirs="$JAVA_ENDORSED_DIRS" -classpath
"$CLASSPATH" \
            -Dcatalina.base="$CATALINA_BASE" \
            -Dcatalina.home="$CATALINA_HOME" \
            -Djava.io.tmpdir="$CATALINA_TMPDIR" \
            org.apache.catalina.startup.Bootstrap "$@" start
    fi
elif [ "$1" = "start" ]; then
    shift
    touch "$CATALINA_BASE"/logs/catalina.out
    if [ "$1" = "-security" ]; then
        echo "Using Security Manager"
        shift
        "$_RUNJAVA" $JAVA_OPTS $CATALINA_OPTS \
            -Djava.endorsed.dirs="$JAVA_ENDORSED_DIRS" -classpath
"$CLASSPATH" \
            -Djava.security.manager \
            -Djava.security.policy=="$CATALINA_BASE"/conf/catalina.policy \
            -Dcatalina.base="$CATALINA_BASE" \
            -Dcatalina.home="$CATALINA_HOME" \
            -Djava.io.tmpdir="$CATALINA_TMPDIR" \
            org.apache.catalina.startup.Bootstrap "$@" start \
            >> "$CATALINA_BASE"/logs/catalina.out 2>&1 &
    else
        if [ ! -z "$CATALINA_PID" ]; then
            echo $! > $CATALINA_PID
        fi
        else
            "$_RUNJAVA" $JAVA_OPTS $CATALINA_OPTS \
                -Djava.endorsed.dirs="$JAVA_ENDORSED_DIRS" -classpath
"$CLASSPATH" \
                -Dcatalina.base="$CATALINA_BASE" \
                -Dcatalina.home="$CATALINA_HOME" \
                -Djava.io.tmpdir="$CATALINA_TMPDIR" \
                org.apache.catalina.startup.Bootstrap "$@" start \
                >> "$CATALINA_BASE"/logs/catalina.out 2>&1 &
        fi
        if [ ! -z "$CATALINA_PID" ]; then
            echo $! > $CATALINA_PID
        fi
    fi
elif [ "$1" = "stop" ]; then

```

```

shift
"$_RUNJAVA" $JAVA_OPTS $CATALINA_OPTS \
-Djava.endorsed.dirs="$JAVA_ENDORSED_DIRS" -classpath "$CLASSPATH" \
-Dcatalina.base="$CATALINA_BASE" \
-Dcatalina.home="$CATALINA_HOME" \
-Djava.io.tmpdir="$CATALINA_TMPDIR" \
org.apache.catalina.startup.Bootstrap "$@" stop

if [ "$1" = "-force" ] ; then
    shift
    if [ ! -z "$CATALINA_PID" ] ; then
        echo "Killing: `cat $CATALINA_PID`"
        kill -9 `cat $CATALINA_PID`
    fi
fi

else
    echo "Usage: catalina.sh ( commands ... )"
    echo "commands:"
    if $os400; then
        echo " debug          Start Catalina in a debugger (not
available on OS400)"
        echo " debug -security    Debug Catalina with a security manager
(not available on OS400)"
    else
        echo " debug          Start Catalina in a debugger"
        echo " debug -security  Debug Catalina with a security manager"
    fi
    echo " embedded      Start Catalina in embedded mode"
    echo " jpdstart       Start Catalina under JPDA debugger"
    echo " run           Start Catalina in the current window"
    echo " run -security  Start in the current window with security
manager"
    echo " start         Start Catalina in a separate window"
    echo " start -security Start in a separate window with security
manager"
    echo " stop         Stop Catalina"
    exit 1
fi

```

### 17.4.3 在 UNIX/Linux 平台上启动 Tomcat

使用 startup.sh 脚本可以方便地启动 Tomcat。startup.sh 脚本会设置正确的环境变量，然后调用 catalina.sh 脚本，并传入参数 start。代码清单 17-13 给出了 startup.sh 脚本的内容。

代码清单 17-13 startup.sh 脚本

```

#!/bin/sh
#
#-----
# Start Script for the CATALINA Server
#
# $Id: startup.sh,v 1.3 2002/08/04 18:19:43 patrickl Exp $
#-----
# resolve links - $0 may be a softlink

```



```

PRG="$0"

while [ -h "$PRG" ] ; do
  ls=`ls -ld "$PRG"`
  link=`expr "$ls" : '.*-> \(.*)$'`
  if expr "$link" : '.*/*' > /dev/null; then
    PRG="$link"
  else
    PRG=`dirname "$PRG"`/"$link"
  fi
done

PRGDIR=`dirname "$PRG"`
EXECUTABLE=catalina.sh

# Check that target executable exists
if [ ! -x "$PRGDIR"/"$EXECUTABLE" ]; then
  echo "Cannot find $PRGDIR/$EXECUTABLE"
  echo "This file is needed to run this program"
  exit 1
fi

exec "$PRGDIR"/"$EXECUTABLE" start "$@"

```

#### 17.4.4 在 UNIX/Linux 平台上关闭 Tomcat

运行 shutdown.sh 脚本可以很方便地关闭 Tomcat。该脚本会调用 catalina.sh 脚本，并传入参数 stop。

代码清单 17-14 给出了 shutdown.sh 脚本的内容。

代码清单 17-14 shutdown.sh 脚本

```

#!/bin/sh
# -----
# Stop script for the CATALINA Server
#
# Id: shutdown.sh,v 1.3 2002/08/04 18:19:43 patrickl Exp $
# -----
# resolve links - $0 may be a softlink
PRG="$0"
while [ -h "$PRG" ] ; do
  ls=`ls -ld "$PRG"`
  link=`expr "$ls" : '.*-> \(.*)$'`
  if expr "$link" : '.*/*' > /dev/null; then
    PRG="$link"
  else
    PRG=`dirname "$PRG"`/"$link"
  fi
done
PRGDIR=`dirname "$PRG"`
EXECUTABLE=catalina.sh

# Check that target executable exists
if [ ! -x "$PRGDIR"/"$EXECUTABLE" ]; then
  echo "Cannot find $PRGDIR/$EXECUTABLE"
  echo "This file is needed to run this program"
  exit 1
fi
exec "$PRGDIR"/"$EXECUTABLE" stop "$@"

```

## 17.5 小结

本章对用来启动应用程序的两个类进行了介绍，分别是 Catalina 类和 Bootstrap 类，它们都位于 org.apache.catalina.startup 包下。此外，还对在 Windows 平台和 UNIX/Linux 平台上用来启动/关闭 Tomcat 的批处理文件和 Shell 脚本进行了介绍。

## 第 18 章

# 部署器

要使用一个 Web 应用程序，必须要将表示该应用程序的 Context 实例部署到一个 Host 实例中。在 Tomcat 中，Context 实例可以用 WAR 文件的形式来部署，也可以将整个 Web 应用程序复制到 Tomcat 安装目录下的 webapp 下。对于部署的每个应用程序，可以在其中包含一个描述符文件，该文件包含 Context 实例的配置信息。描述符文件采用 XML 文档格式。

**注意** 在 Tomcat 4 和 Tomcat 5 中，使用了两个应用程序来管理 Tomcat 和部署在 Tomcat 中的 Web 应用程序，分别是 manager 应用程序和 admin 应用程序。这两个应用程序所使用到的类文件都位于 %CATALINA\_HOME%/server/webapps 目录下，并且分别使用了两个描述符文件：manager.xml 和 admin.xml。在 Tomcat 4 中，在 %CATALINA\_HOME%/webapps 目录下，有 3 个描述符文件；在 Tomcat 5 中，它们在相关应用程序的目录中，即，分别位于 %CATALINA\_HOME%/server/webapps/admin 目录和 %CATALINA\_HOME%/server/webapps/manager 目录下。

本章将会讨论如何使用部署器来部署一个 Web 应用程序。在 Tomcat 中，部署器是 org.apache.catalina.Deployer 接口的实例。部署器与一个 Host 实例相关联，用来安装 Context 实例。安装 Context 实例的意思是，创建一个 StandardContext 实例，并将该实例添加到 Host 实例中。创建的 Context 实例会随其父容器——Host 实例一起启动（因此，除了 Wrapper 实例类，容器的实例总是会调用其子容器的 start() 方法）。但是，部署器也可以用来单独地启动和关闭 Context 实例。

在本章中，你首先会学习到 Tomcat 如何在 Host 实例中部署一个 Web 应用程序。然后，将对 Deployer 接口及其标准实现类（org.apache.catalina.core.StandardHostDeployer 类）的工作原理进行介绍。

### 18.1 部署一个 Web 应用程序

在 15 章使用如下代码实例化 StandardHost 类，并将一个 Context 实例作为子容器添加到 Host 实例中：

```
Context context = new StandardContext();
context.setPath("/app1");
context.setDocBase("app1");
LifecycleListener listener = new ContextConfig();
((Lifecycle) context).addLifecycleListener(listener);
```

```
Host host = new StandardHost();
host.addChild(context);
```



这是之前部署应用程序的方法。但是，在 Tomcat 中并没有这样部署应用程序。那在实际部署环境中 Context 实例如何被添加到 Host 容器中的呢？答案在于 StandardHost 实例中使用的 org.apache.catalina.startup.HostConfig 类型的生命周期监听器。

当调用 StandardHost 实例的 start() 方法时，它会触发 START 事件。HostConfig 实例会对此事件进行相应，并调用其自身的 start() 方法，该方法会逐个部署并安装指定目录中的所有 Web 应用程序。下面对其中的细节进行讲解。

回忆一下第 15 章如何使用 Digester 对象来解析 XML 文档的内容。但是，它并没有涉及 Digester 对象中所有的规则。其中被忽略掉的一个主题就是部署器，也就是本章的主题。

在 Tomcat 中，org.apache.catalina.startup.Catalina 类是启动类，使用 Digester 对象来解析 server.xml 文件，将其中的 XML 元素转换为 Java 对象。Catalina 类定义了 createStartDigester() 方法来为 Digester 对象添加规则。createStartDigester() 方法中的其中一行如下所示：

```
digester.addRuleSet(new HostRuleSet("Server/Service/Engine/"));
```

org.apache.catalina.startup.HostRuleSet 类继承自 org.apache.commons.digester.RuleSetBase 类（该类在第 15 章中介绍过）。作为 RuleSetBase 类的一个子类，HostRuleSet 类必须提供 addRuleInstances() 方法的实现，需要在该方法中定义 RuleSet 的规则。下面是 HostRuleSet 类中 addRuleInstances() 方法的片段：

```
public void addRuleInstances(Digester digester) {
    digester.addObjectCreate(prefix + "Host",
        "org.apache.catalina.core.StandardHost", "className");
    digester.addSetProperties(prefix + "Host");
    digester.addRule(prefix + "Host",
        new CopyParentClassLoaderRule(digester));
    digester.addRule(prefix + "Host",
        new LifecycleListenerRule(digester,
            "org.apache.catalina.startup.HostConfig", "hostConfigClass"));
}
```

正如上面代码所示，当在 server.xml 文件中遇到符合“Server/Service/Engine/Host”模式的标签时，会创建 org.apache.catalina.startup.HostConfig 类的一个实例，并将其添加到 Host 实例中，作为生命周期监听器。换句话说，HostConfig 类会处理 StandardHost 实例的 start() 方法和 stop() 触发的事件。

代码清单 18-1 给出了 HostConfig 类的 lifecycleEvent() 方法的实现。该方法是一个事件处理程序。因为 HostConfig 的实例是 StandardHost 实例的监听器，每当 StandardHost 实例启动或关闭时，都会调用 lifecycleEvent() 方法。

代码清单 18-1 HostConfig 类的 lifecycleEvent() 方法的实现

```
public void lifecycleEvent(LifecycleEvent event) {
    // Identify the host we are associated with
    try {
        host = (Host) event.getLifecycle();
        if (host instanceof StandardHost) {
            int hostDebug = ((StandardHost) host).getDebug();
            if (hostDebug > this.debug) {
                this.debug = hostDebug;
            }
            setDeployXML(((StandardHost) host).isDeployXML());
            setLiveDeploy(((StandardHost) host).getLiveDeploy());
        }
    }
}
```

```

        setUnpackWARs(((StandardHost) host).isUnpackWARs());
    }
}
catch (ClassCastException e) {
    log(sm.getString("hostConfig.cce", event.getLifecycle()), e);
    return;
}
// Process the event that has occurred
if (event.getType().equals(Lifecycle.START_EVENT))
    start();
else if (event.getType().equals(Lifecycle.STOP_EVENT))
    stop();
}

```

如果变量 `host` 指向的对象是 `org.apache.catalina.core.StandardHost` 类的实例，就会调用 `setDeployXML()` 方法、`setLiveDeploy()` 方法和 `setUnpackWARs()` 方法：

```

setDeployXML(((StandardHost) host).isDeployXML());
setLiveDeploy(((StandardHost) host).getLiveDeploy());
setUnpackWARs(((StandardHost) host).isUnpackWARs());

```

`StandardHost` 类的 `isDeployXML()` 方法指明了 `Host` 实例是否要部署一个 `Context` 实例的描述符文件。默认情况下，`deployXML` 属性的值为 `true`。`liveDeploy` 属性指明了 `Host` 实例是否要周期性地检查一个新的部署，`unpackWARs` 属性指明是要将 `WAR` 文件形式的 `Web` 应用程序解压缩。

在接收到 `START` 事件通知后，`HostConfig` 对象的 `lifecycleEvent()` 方法会调用 `start()` 方法来部署 `Web` 应用程序。代码清单 18-2 给出了 `start()` 方法的实现。

代码清单 18-2 `HostConfig` 类的 `start()` 方法的实现

```

protected void start() {
    if (debug >= 1)
        log(sm.getString("hostConfig.start"));
    if (host.getAutoDeploy()) {
        deployApps();
    }
    if (isLiveDeploy()) {
        threadStart();
    }
}

```

当 `autoDeploy` 属性的值为 `true` 时（默认情况下，该值为 `true`），`start()` 方法会调用 `deployApps()` 方法。此外，若 `liveDeploy` 属性的值为 `true`（默认情况下，该值为 `true`），它还会调用 `threadStart()` 方法来派生一个新线程。动态部署将在 18.1.4 节讨论。

`deployApps()` 方法会获取 `Host` 实例的 `appBase` 属性的值，默认为 `webapps` 的值（参见 `Tomcat` 的 `server.xml` 文件内容）。部署进程会将 `%CATALINE_HOME%/webapps` 目录下的所有目录都看做为 `Web` 应用程序的目录来执行部署工作。此外，该目录中所有的 `WAR` 文件和描述符文件也都会进行部署。

代码清单 18-3 给出了 `deployApps()` 方法的实现。

代码清单 18-3 deployApps() 方法的实现

```
protected void deployApps() {
    if (!host instanceof Deployer)
        return;
    if (debug >= 1)
        log(sm.getString("hostConfig.deploying"));
    File appBase = appBase();
    if (!appBase.exists() || !appBase.isDirectory())
        return;
    String files[] = appBase.list();
    deployDescriptors(appBase, files);
    deployWARs(appBase, files);
    deployDirectories(appBase, files);
}
```

deployApps() 方法会调用其他 3 个方法：deployDescriptors()、deployWARs() 和 deployDirectories()。deployApps() 方法会将 File 类型的变量 appBase 和 webapps 目录下所有的文件的数组形式传递给这 3 个方法。Context 实例是通过它的路径来标识的，每个部署的 Context 实例都必须有一条唯一的路径。已经部署的 Context 实例会被添加到 HostConfig 对象中已部署的 ArrayList 中。因此，在部署一个 Context 实例之前，deployDescriptors() 方法、deployWARs() 方法和 deployDirectories() 方法必须保证已部署的 ArrayList 没有具有相同路径的 Context 实例。

下面来逐个看一下这 3 个部署方法是如何运行的。在阅读了下面的 3 节后，你应该已经可以回答“这 3 个方法的调用顺序重要吗？”这个问题（当然，答案是肯定的）。

### 18.1.1 部署一个描述符

可以编写一个 XML 文件来描述 Context 对象。例如，在 Tomcat 4 和 5 中的 admin 和 manager 应用中就分别使用代码清单 18-4 和代码清单 18-5 中的两个 XML 文件。

代码清单 18-4 admin 应用程序的描述符文件 (admin.xml)

```
<Context path="/admin" docBase="../server/webapps/admin"
  debug="0" privileged="true">
  <!-- Uncomment this Valve to limit access to the Admin app to
    localhost for obvious security reasons. Allow may be a comma-
    separated list of hosts (or even regular expressions).
  <Valve className="org.apache.catalina.valves.RemoteAddrValve"
    allow="127.0.0.1"/>
  -->
  <Logger className="org.apache.catalina.logger.FileLogger"
    prefix="localhost_admin_log." suffix=".txt"
    timestamp="true"/>
</Context>
```

代码清单 18-5 manager 应用程序中的描述符文件 (manager.xml)

```
<Context path="/manager" docBase="../server/webapps/manager"
  debug="0" privileged="true">
  <!-- Link to the user database we will get roles from -->
  <ResourceLink name="users" global="UserDatabase"
    type="org.apache.catalina.UserDatabase"/>
```



```
</Context>
```

注意，这两个描述符都有一个 Context 元素。Context 元素中的 docBase 属性的值分别为 %CATALINA\_HOME%/server/webapps/admin 和 %CATALINA\_HOME%/server/webapps/manager，这表明，admin 应用程序和 manager 应用程序并没有部署到默认的地方。

HostConfig 类使用了代码清单 18-6 中给出了 deployDescriptors() 方法来部署 XML 文件。在 Tomcat 4 中，这些文件位于 %CATALINA\_HOME%/webapps 目录下；在 Tomcat 5 中，位于 %CATALINA\_HOME%/server/webapps/ 目录的子目录下。

代码清单 18-6 HostConfig 类的 deployDescriptors() 方法的实现

```
protected void deployDescriptors(File appBase, String[] files) {
    if (!deployXML)
        return;

    for (int i = 0; i < files.length; i++) {
        if (files[i].equalsIgnoreCase("META-INF"))
            continue;
        if (files[i].equalsIgnoreCase("WEB-INF"))
            continue;
        if (deployed.contains(files[i]))
            continue;
        File dir = new File(appBase, files[i]);
        if (files[i].toLowerCase().endsWith(".xml")) {
            deployed.add(files[i]);

            // Calculate the context path and make sure it is unique
            String file = files[i].substring(0, files[i].length() - 4);
            String contextPath = "/" + file;
            if (file.equals("ROOT")) {
                contextPath = "";
            }
            if (host.findChild(contextPath) != null) {
                continue;
            }

            // Assume this is a configuration descriptor and deploy it
            log(sm.getString("hostConfig.deployDescriptor", files[i]));
            try {
                URL config =
                    new URL("file", null, dir.getCanonicalPath());
                ((Deployer) host).install(config, null);
            }
            catch (Throwable t) {
                log(sm.getString("hostConfig.deployDescriptor.error",
                    files[i]), t);
            }
        }
    }
}
```

## 18.1.2 部署一个 WAR 文件

可以将 Web 应用程序以一个 WAR 形式的文件来部署。HostConfig 类使用代码清单 18-7 中

给出的 `deployWARs()` 方法，将位于 `%CATALINA_HOME%/webapps` 目录下的任何 WAR 文件进行部署。

代码清单 18-7 HostConfig 类的 `deployWARs()` 方法的实现

```
protected void deployWARs(File appBase, String[] files) {
    for (int i = 0; i < files.length; i++) {
        if (files[i].equalsIgnoreCase("META-INF"))
            continue;
        if (files[i].equalsIgnoreCase("WEB-INF"))
            continue;
        if (deployed.contains(files[i]))
            continue;
        File dir = new File(appBase, files[i]);

        if (files[i].toLowerCase().endsWith(".war")) {
            deployed.add(files[i]);
            // Calculate the context path and make sure it is unique
            String contextPath = "/" + files[i];
            int period = contextPath.lastIndexOf(".");
            if (period >= 0)
                contextPath = contextPath.substring(0, period);
            if (contextPath.equals("/ROOT"))
                contextPath = "";
            if (host.findChild(contextPath) != null)
                continue;

            if (isUnpackWARs()) {
                // Expand and deploy this application as a directory
                log(sm.getString("hostConfig.expand", files[i]));
                try {
                    URL url = new URL("jar:file:" +
                        dir.getCanonicalPath() + "!/");
                    String path = expand(url);
                    url = new URL("file:" + path);
                    ((Deployer) host).install(contextPath, url);
                } catch (Throwable t) {
                    log(sm.getString("hostConfig.expand.error", files[i]), t);
                }
            } else {
                // Deploy the application in this WAR file
                log(sm.getString("hostConfig.deployJar", files[i]));
                try {
                    URL url = new URL("file", null,
                        dir.getCanonicalPath());
                    url = new URL("jar:" + url.toString() + "!/");
                    ((Deployer) host).install(contextPath, url);
                } catch (Throwable t) {
                    log(sm.getString("hostConfig.deployJar.error",
                        files[i]), t);
                }
            }
        }
    }
}
```

### 18.1.3 部署一个目录

也可以直接将 Web 应用程序的整个目录复制到 %CATALINA\_HOME%/webapps 目录下来完成 Web 应用程序的部署。HostConfig 类使用代码清单 18-8 中给出的 deployDirectories() 方法完成对这些 Web 应用程序的部署。

代码清单 18-8 HostConfig 类的 deployDirectories() 方法的实现

```
protected void deployDirectories (File appBase, string[] files){
    for (int i = 0; i < files.length; i++) {
        if (files[i].equalsIgnoreCase("META-INF"))
            continue;
        if (files[i].equalsIgnoreCase("WEB-INF"))
            continue;
        if (deployed.contains(files[i]))
            continue;
        File dir = new File(appBase, files[i]);
        if (dir.isDirectory()) {
            deployed.add(files[i]);

            // Make sure there is an application configuration directory
            // This is needed if the Context appBase is the same as the
            // web server document root to make sure only web applications
            // are deployed and not directories for web space.
            File webInf = new File(dir, "/WEB-INF");
            if (!webInf.exists() || !webInf.isDirectory() ||
                !webInf.canRead())
                continue;

            // Calculate the context path and make sure it is unique
            String contextPath = "/" + files[i];
            if (files[i].equals("ROOT"))
                contextPath = "";
            if (host.findChild(contextPath) != null)
                continue;

            // Deploy the application in this directory
            log(sm.getString("hostConfig.deployDir", files[i]));
            try {
                URL url = new URL("file", null, dir.getCanonicalPath());
                ((Deployer) host).install(contextPath, url);
            }
            catch (Throwable t) {
                log(sm.getString("hostConfig.deployDir.error", files[i]), t);
            }
        }
    }
}
```

### 18.1.4 动态部署

正如前面提到的，StandardHost 实例使用 HostConfig 对象作为一个生命周期监听器。当 StandardHost 对象启动时，它的 start() 方法会触发一个 START 事件。为了响应 START 事件，HostConfig 中的 LifecycleEvent() 方法和 HostConfig 中的事件处理程序调用 start() 方法。在



Tomcat 4 中, 在 start() 方法的最后一行, 当 liveDeploy 属性为 true 时 (默认情况下, 该属性为 true), start() 方法会调用 threadStart() 方法:

```
if (isLiveDeploy()) {
    threadStart();
}
```

threadStart() 方法会派生一个新线程并调用 run() 方法。run() 方法会定期检查是否有新应用要部署, 或已经部署的 Web 应用程序的 web.xml 是否有修改。代码清单 18-9 给出了 run() 方法的实现。

代码清单 18-9 Tomcat 4 中 HostConfig 类的 run() 方法的实现

```
/**
 * The background thread that checks for web application autoDeploy
 * and changes to the web.xml config.
 */
public void run() {
    if (debug >= 1)
        log("BACKGROUND THREAD Starting");
    // Loop until the termination semaphore is set
    while (!threadDone) {
        // Wait for our check interval
        threadSleep();
        // Deploy apps if the Host allows auto deploying
        deployApps();
        // Check for web.xml modification
        checkWebXmlLastModified();
    }
    if (debug >= 1)
        log("BACKGROUND THREAD Stopping");
}
```

threadSleep() 方法会使该线程休眠一段时间, 该时间的长度由属性 checkInterval 指定, 该属性值默认为 15 秒, 即每隔 15 秒检查一次。

在 Tomcat 5 中, HostConfig 类没有再使用专用线程来执行检查工作, 而是由 StandardHost 类的 backgroundProcess() 方法周期性地触发一个 "check" 事件:

```
public void backgroundProcess() {
    lifecycle.fireLifecycleEvent("check", null);
}
```

**注意** backgroundProcess() 方法会由一个专门的线程来周期性地调用, 用来执行容器中所有的后台处理工作。

接收到 check 事件后, 生命周期监听器 (即 HostConfig 对象) 会调用其 check() 方法执行部署应用的检查:

```
public void lifecycleEvent(LifecycleEvent event) {
    if (event.getType().equals("check"))
        check();
}
```

代码清单 18-10 给出了 Tomcat 5 中 HostConfig 类的 check() 方法的实现。

代码清单 18-10 Tomcat 5 中 HostConfig 类的 check() 方法的实现

```
protected void check() {
    if (host.getAutoDeploy()) {
        // Deploy apps if the Host allows auto deploying
        deployApps();
        // Check for web.xml modification
        checkContextLastModified();
    }
}
```

如你所见, check() 方法会调用 deployApps() 方法。在 Tomcat 4 和 Tomcat 5 中, deployApps() 方法都会完成 Web 应用程序的部署工作。该方法的实现已经在代码清单 18-3 中给出。正如前面讨论过的, 该方法会调用 deployDescriptors() 方法、deployWARs() 方法和 deployDirectories() 方法。

在 Tomcat 5 中, check() 方法还会调用 checkContextLastModified() 方法, 后者遍历所有已经部署的 Context, 检查 web.xml 文件的时间戳, 及每个 Context 中 WEB-INF 目录下的内容。如果某个检查的资源被修改了, 会重新启动相应的 Context 实例。此外, checkContextLastModified() 方法还会检查所有已经部署的 WAR 文件的时间戳, 如果某个应用程序的 WAR 文件被修改了, 会重新对该应用程序进行部署。

在 Tomcat 4 中, 后台线程的 run() 方法会调用 checkWebXmlLastModified() 方法来完成与 Tomcat 5 中 checkContextLastModified() 方法类似的工作。

## 18.2 Deploy 接口

部署器是 org.apache.catalina.Deployer 接口的实例。StandardHost 类实现 Deployer 接口。因此, StandardHost 实例也是一个部署器, 它是一个容器, Web 应用可以部署到其中, 或从其中取消部署。代码清单 18-11 给出了 Deployer 接口的定义。

代码清单 18-11 Deployer 接口的定义

```
package org.apache.catalina;

import java.io.IOException;
import java.net.URL;

/**
 * A <b>Deployer</b> is a specialized Container into which web
 * applications can be deployed and undeployed. Such a Container
 * will create and install child Context instances for each deployed
 * application. The unique key for each web application will be the
 * context path to which it is attached.
 *
 * @author Craig R. McClanahan
 * @version $Revision: 1.6 $ $Date: 2002/04/09 23:48:21 $
 */
public interface Deployer {
    /**
     * The ContainerEvent event type sent when a new application is
     * being installed by <code>install()</code>, before it has been

```

```

    * started.
    */
    public static final String PRE_INSTALL_EVENT = "pre-install";

    /**
     * The ContainerEvent event type sent when a new application is
     * installed by <code>install()</code>, after it has been started.
     */
    public static final String INSTALL_EVENT = "install";

    /**
     * The ContainerEvent event type sent when an existing application is
     * removed by <code>remove()</code>.
     */
    public static final String REMOVE_EVENT = "remove";

    /**
     * Return the name of the Container with which this Deployer is
     * associated.
     */
    public String getName();

    /**
     * Install a new web application, whose web application archive is at
     * the specified URL, into this container with the specified context.
     * path. A context path of "" (the empty string) should be used for
     * the root application for this container. Otherwise, the context
     * path must start with a slash.
     * <p>
     * If this application is successfully installed, a ContainerEvent of
     * type <code>INSTALL_EVENT</code> will be sent to all registered
     * listeners,
     * with the newly created <code>Context</code> as an argument.
     *
     * @param contextPath The context path to which this application
     * should be installed (must be unique)
     * @param war A URL of type "jar:" that points to a WAR file, or type
     * "file:" that points to an unpacked directory structure containing
     * the web application to be installed
     *
     * @exception IllegalArgumentException if the specified context path
     * is malformed (it must be "" or start with a slash)
     * @exception IllegalStateException if the specified context path
     * is already attached to an existing web application
     * @exception IOException if an input/output error was encountered
     * during installation
     */
    public void install(String contextPath, URL war) throws IOException;

    /**
     * <p>Install a new web application, whose context configuration file
     * (consisting of a <code><Context></code> element) and web
     * application archive are at the specified URLs.</p>
     *
     * <p>If this application is successfully installed, a ContainerEvent
     * of type <code>INSTALL_EVENT</code> will be sent to all registered
     * listeners, with the newly created <code>Context</code> as an
     * argument.
     * </p>
     *
     * @param config A URL that points to the context configuration file
     * to be used for configuring the new Context
     * @param war A URL of type "jar:" that points to a WAR file, or type

```



```

* "file:" that points to an unpacked directory structure containing
* the web application to be installed
*
* @exception IllegalArgumentException if one of the specified URLs
* is null
* @exception IllegalStateException if the context path specified in
* the context configuration file is already attached to an existing
* web application
* @exception IOException if an input/output error was encountered
* during installation
*/
public void install(URL config, URL war) throws IOException;

/**
* Return the Context for the deployed application that is associated
* with the specified context path (if any); otherwise return
* <code>null</code>.
*
* @param contextPath The context path of the requested web
* application
*/
public Context findDeployedApp(String contextPath);

/**
* Return the context paths of all deployed web applications in this
* Container. If there are no deployed applications, a zero-length
* array is returned.
*/
public String[] findDeployedApps();

/**
* Remove an existing web application, attached to the specified
* context path. If this application is successfully removed, a
* ContainerEvent of type <code>REMOVE_EVENT</code> will be sent to
* all registered listeners, with the removed <code>Context</code> as
* an argument.
*
* @param contextPath The context path of the application to be
* removed
*
* @exception IllegalArgumentException if the specified context path
* is malformed (it must be "" or start with a slash)
* @exception IllegalArgumentException if the specified context path
* does not identify a currently installed web application
* @exception IOException if an input/output error occurs during
* removal
*/
public void remove(String contextPath) throws IOException;

/**
* Start an existing web application, attached to the specified
* context path. Only starts a web application if it is not running.
*
* @param contextPath The context path of the application to be
* started
*
* @exception IllegalArgumentException if the specified context path
* is malformed (it must be "" or start with a slash)
* @exception IllegalArgumentException if the specified context path
* does not identify a currently installed web application
* @exception IOException if an input/output error occurs during
* startup
*/

```

```

public void start(String contextPath) throws IOException;

/**
 * Stop an existing web application, attached to the specified
 * context path. Only stops a web application if it is running.
 *
 * @param contextPath The context path of the application to be
 * stopped
 * @exception IllegalArgumentException if the specified context path
 * is malformed (it must be "" or start with a slash)
 * @exception IllegalArgumentException if the specified context path
 * does not identify a currently installed web application
 * @exception IOException if an input/output error occurs while
 * stopping the web application
 */
public void stop(String contextPath) throws IOException;
}

```

StandardHost 类使用一个辅助类 (org.apache.catalina.core.StandardHostDeployer) 来完成部署与安装 Web 应用程序的相关任务。下面的代码片段展示了 StandardHost 对象如何将部署任务委托给 StandardHostDeployer 实例来完成:

```

/**
 * The <code>Deployer</code> to whom we delegate application
 * deployment requests.
 */
private Deployer deployer = new StandardHostDeployer(this);

public void install(String contextPath, URL war) throws IOException {
    deployer.install(contextPath, war);
}

public synchronized void install(URL config, URL war) throws
IOException {
    deployer.install(config, war);
}

public Context findDeployedApp(String contextPath) {
    return (deployer.findDeployedApp(contextPath));
}

public String[] findDeployedApps() {
    return (deployer.findDeployedApps());
}

public void remove(String contextPath) throws IOException {
    deployer.remove(contextPath);
}

public void start(String contextPath) throws IOException {
    deployer.start(contextPath);
}

public void stop(String contextPath) throws IOException {
    deployer.stop(contextPath);
}
}

```

StandardHostDeployer 类将在 18.3 节介绍。

## 18.3 StandardHostDeployer 类

org.apache.catalina.core.StandardHostDeployer 类是一个辅助类, 帮助完成将 Web 应用程序部

署到 StandardHost 实例的工作。StandardHostDeployer 实例由 StandardHost 对象调用，在其构造函数中，会传入 StandardHostDeployer 类的一个实例：

```
public StandardHostDeployer(StandardHost host) {
    super();
    this.host = host;
}
```

下面几节将对 StandardHostDeployer 类的几个方法进行介绍。

### 18.3.1 安装一个描述符

StandardHostDeployer 类有两个 install() 方法，分别在本节和下一节来讨论，本节讨论的 install() 方法用于安装一个描述符，下一节将要讨论的 install() 方法用来安装一个 WAR 文件或一个目录。

代码清单 18-12 给出的 install() 方法用来安装描述符。当 HostConfig 对象的 deployDescriptors() 方法调用了其 install() 方法后，StandardHost 实例调用该 install() 方法。

代码清单 18-12 用来安装描述符的 install() 方法的实现

```
public synchronized void install(URL config, URL war)
    throws IOException {
    // Validate the format and state of our arguments
    if (config == null)
        throw new IllegalArgumentException(
            (sm.getString("standardHost.configRequired")));
    if (!host.isDeployXML())
        throw new IllegalArgumentException(
            (sm.getString("standardHost.configNotAllowed")));
    // Calculate the document base for the new web application (if
    // needed)
    String docBase = null; // Optional override for value in config file
    if (war != null) {
        String url = war.toString();
        host.log(sm.getString("standardHost.installingWAR", url));
        // Calculate the WAR file absolute pathname
        if (url.startsWith("jar:")) {
            url = url.substring(4, url.length() - 2);
        }
        if (url.startsWith("file://"))
            docBase = url.substring(7);
        else if (url.startsWith("file:"))
            docBase = url.substring(5);
        else
            throw new IllegalArgumentException(
                (sm.getString("standardHost.warURL", url)));
    }
    // Install the new web application
    this.context = null;
    this.overrideDocBase = docBase;
    InputStream stream = null;
    try {
        stream = config.openStream();
        Digester digester = createDigester();
        digester.setDebug(host.getDebug());
        digester.clear();
```



```

        digester.push(this);
        digester.parse(stream);
        stream.close();
        stream = null;
    }
    catch (Exception e) {
        host.log
            (sm.getString("standardHost.installError", docBase), e);
        throw new IOException(e.toString());
    }
    finally {
        if (stream != null) {
            try {
                stream.close();
            }
            catch (Throwable t) {
                ;
            }
        }
    }
}

```

### 18.3.2 安装一个 WAR 文件或目录

第二个 `install()` 方法接受一个表示上下文路径的字符串和一个表示 WAR 文件的 URL。代码清单 18-13 给出了该 `install()` 方法的实现。

代码清单 18-13 用于安装 WAR 文件或目录的 `install()` 方法的实现

```

public synchronized void install(String contextPath, URL war)
    throws IOException {
    // Validate the format and state of our arguments
    if (contextPath == null)
        throw new IllegalArgumentException
            (sm.getString("standardHost.pathRequired"));
    if (!contextPath.equals("") && !contextPath.startsWith("/"))
        throw new IllegalArgumentException
            (sm.getString("standardHost.pathFormat", contextPath));
    if (findDeployedApp(contextPath) != null)
        throw new IllegalStateException
            (sm.getString("standardHost.pathUsed", contextPath));
    if (war == null)
        throw new IllegalArgumentException
            (sm.getString("standardHost.warRequired"));

    // Calculate the document base for the new web application
    host.log(sm.getString("standardHost.installing",
        contextPath, war.toString()));
    String url = war.toString();
    String docBase = null;
    if (url.startsWith("jar:")) {
        url = url.substring(4, url.length() - 2);
    }
    if (url.startsWith("file://"))
        docBase = url.substring(7);
    else if (url.startsWith("file:"))
        docBase = url.substring(5);
    else

```

```

throw new IllegalArgumentException
    (sm.getString("standardHost.warURL", url));
// Install the new web application
try {
    Class clazz = Class.forName(host.getContextClass());
    Context context = (Context) clazz.newInstance();
    context.setPath(contextPath);

    context.setDocBase(docBase);
    if (context instanceof Lifecycle) {
        clazz = Class.forName(host.getConfigClass());
        LifecycleListener listener =
            (LifecycleListener) clazz.newInstance();
        ((Lifecycle) context).addLifecycleListener(listener);
    }
    host.fireContainerEvent(PRE_INSTALL_EVENT, context);
    host.addChild(context);
    host.fireContainerEvent(INSTALL_EVENT, context);
} catch (Exception e) {
    host.log(sm.getString("standardHost.installError", contextPath),
        e);
    throw new IOException(e.toString());
}
}

```

注意，当安装一个 Context 后，就会将其添加到 StandardHost 实例中。

### 18.3.3 启动 Context 实例

StandardHostDeployer 类中的 start() 方法用于启动 Context 实例。代码清单 18-14 给出了 start() 方法的实现。

代码清单 18-14 StandardHostDeployer 类的 start() 方法的实现

```

public void start(String contextPath) throws IOException {
    // Validate the format and state of our arguments
    if (contextPath == null)
        throw new IllegalArgumentException
            (sm.getString("standardHost.pathRequired"));
    if (!contextPath.equals("") && !contextPath.startsWith("/"))
        throw new IllegalArgumentException
            (sm.getString("standardHost.pathFormat", contextPath));
    Context context = findDeployedApp(contextPath);
    if (context == null)
        throw new IllegalArgumentException
            (sm.getString("standardHost.pathMissing", contextPath));
    host.log("standardHost.start " + contextPath);
    try {
        ((Lifecycle) context).start();
    } catch (LifecycleException e) {
        host.log("standardHost.start " + contextPath + ": ", e);
        throw new IllegalStateException
            ("standardHost.start " + contextPath + ": " + e);
    }
}
}

```

### 18.3.4 停止一个 Context 实例

StandardHostDeployer 类的 stop() 方法可以用来停止一个 Context 实例。代码清单 18-15 中给出了 stop() 方法的实现。

代码清单 18-15 StandardHostDeployer 类中 stop() 方法的实现

```
public void stop(String contextPath) throws IOException {
    // Validate the format and state of our arguments
    if (contextPath == null)
        throw new IllegalArgumentException(
            sm.getString("standardHost.pathRequired"));
    if (!contextPath.equals("") && !contextPath.startsWith("/"))
        throw new IllegalArgumentException(
            sm.getString("standardHost.pathFormat", contextPath));
    Context context = findDeployedApp(contextPath);
    if (context == null)
        throw new IllegalArgumentException(
            sm.getString("standardHost.pathMissing", contextPath));
    host.log("standardHost.stop " + contextPath);
    try {
        ((Lifecycle) context).stop();
    }
    catch (LifecycleException e) {
        host.log("standardHost.stop " + contextPath + ": ", e);
        throw new IllegalStateException(
            "standardHost.stop " + contextPath + ": " + e);
    }
}
```

## 18.4 小结

部署器是用来部署和安装 Web 应用程序的组件，是 org.apache.catalina.Deployer 接口的实例。StandardHost 类实现 Deployer 接口，使其成为一个可以向其中部署 Web 应用程序的特殊容器。StandardHost 类会将部署和安装 Web 应用程序的任务委托给其辅助类 org.apache.catalina.core.StandardHostDeployer 类完成。StandardHostDeployer 类提供了部署和安装 Web 应用程序，以及启动 / 关闭 Context 实例的代码。



## 第 19 章

# Manager 应用程序的 servlet 类

Tomcat 4 和 Tomcat 5 附带了 Manager 应用程序，可以用来管理已经部署的 Web 应用程序。与其他的应用程序不同，Manager 应用程序并不在默认的应用程序部署目录 %CATALINA\_HOME%/webapps 中，而是在 %CATALINA\_HOME%/server/webapps 目录中。Manager 应用程序会随 Tomcat 一起启动。因为 Manager 应用程序使用一个描述符文件 manager.xml 来完成部署。在 Tomcat 4 中，该文件位于 %CATALINA\_HOME%/webapps 目录中，在 Tomcat 5 中，该文件位于 %CATALINA\_HOME%/server/webapps 目录中。

**注意** 有关于 Context 描述符的内容已经在第 18 章中介绍过了。

本章主要介绍 Manager 应用程序。首先会介绍如何使用 Manager 应用程序，使你了解 Manager 应用程序是如何运行的。然后，再对 ContainerServlet 接口进行介绍。

### 19.1 使用 Manager 应用程序

在 Tomcat 4 和 Tomcat 5 中，Manager 应用程序都位于 %CATALINA\_HOME%/server/webapps/manager 目录下。该应用程序中的主 servlet 类是 ManagerServlet 类。在 Tomcat 4 中，该类位于 org.apache.catalina.servlets 包下。而在 Tomcat 5 中，该类位于 org.apache.catalina.manager 下，是作为 WEB-INF/lib 目录下的一个 JAR 文件部署的。

**注意** 相对于 Tomcat 5，Tomcat 4 中的 Manager 应用程序更简单一些，但更适合于学习。

因此，本章以 Tomcat 4 中的 Manager 应用程序为例进行介绍。在学习了本章内容后，你应该也可以理解 Tomcat 5 中的 Manager 应用程序。

下面是 Tomcat 4 中 Manager 应用程序的部署描述符中关于 servlet 的定义：

```
<servlet>
  <servlet-name>Manager</servlet-name>
  <servlet-class>
    org.apache.catalina.servlets.ManagerServlet
  </servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>2</param-value>
  </init-param>
</servlet>
<servlet>
  <servlet-name>HTMLManager</servlet-name>
  <servlet-class>
    org.apache.catalina.servlets.HTMLManagerServlet
  </servlet-class>
```

```

<init-param>
  <param-name>debug</param-name>
  <param-value>2</param-value>
</init-param>
</servlet>

```

上面定义的第 1 个 servlet 类是 `org.apache.catalina.servlets.ManagerServlet` 类, 第 2 个 servlet 类是 `org.apache.catalina.servlets.HTMLManagerServlet` 类。本章将着重于介绍 `ManagerServlet` 类。

该应用程序的描述符是 `manager.xml`, 指定了该 Web 应用程序所对应的上下文路径为 “/manager”:

```

<Context path="/manager" docBase="../server/webapps/manager"
  debug="0" privileged="true">
  <!-- Link to the user database we will get roles from -->
  <ResourceLink name="users" global="UserDatabase"
    type="org.apache.catalina.UserDatabase"/>
</Context>

```

第 1 个 `servlet-mapping` 元素指明了如何调用 `ManagerServlet` 类:

```

<servlet-mapping>
  <servlet-name>Manager</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>

```

换句话说, 以下面的模式开始的 URL 可以调用 `ManagerServlet`:

`http://localhost:8080/manager/`

但是, 值得注意的是, 在该部署描述符中, 还有 `security-constraint` 元素:

```

<security-constraint>
  <web-resource-collection>
    <web-resource-name>Entire Application</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <!-- NOTE: This role is not present in the default users file -->
    <role-name>manager</role-name>
  </auth-constraint>
</security-constraint>

```

上面的配置的意思是, 整个应用程序被限制为只有拥有 `manager` 角色的用户才能访问。 `auth-login` 元素指明了权限认证使用 BASIC 身份验证方式, 只有当用户输入正确的用户名和密码时才能访问受限资源:

```

<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Tomcat Manager Application</realm-name>
</login-config>

```

在 Tomcat 中, 用户和角色列表存储于 `tomcat-users.xml` 文件中, 该文件位于 `%CATALINA_HOME%/conf` 目录下。因此, 要访问 Manager 应用程序, 必须添加一个 `manager` 角色和一个拥有该角色的用户, 例如:

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="manager"/>
  <user username="tomcat" password="tomcat" roles="manager" />
</tomcat-users>
```

通过这个 tomcat-users.xml 文件，就可以以 tomcat 为用户名、tomcat 为密码来访问 Manager 应用程序了。

下面列出了 ManagerServlet 类中的一些函数：

- list()
- start()
- stop()
- reload()
- remove()
- resources()
- roles()
- sessions()
- undeploy()

查看该类的 doGet() 方法可以看到如何调用相应的函数。

## 19.2 Containerservlet 接口

实现了 org.apache.catalina.ContainerServlet 接口的 servlet 类可以访问表示该 servlet 实例的 StandardWrapper 对象。通过访问该 Wrapper 实例，它也就可以访问表示当前 Web 应用程序的 Context 实例，以及该 Context 实例内的部署器（StandardHost 类的实例）等对象。

代码清单 19-1 给出了 ContainerServlet 接口的定义。

代码清单 19-1 ContainerServlet 接口的定义

```
package org.apache.catalina;

public interface ContainerServlet {
    public Wrapper getWrapper();
    public void setWrapper(Wrapper wrapper);
}
```

Catalina 实例会调用实现了 ContainerServlet 接口的 servlet 类的 setWrapper() 方法，将该引用传递给表示该 servlet 类的 StandardWrapper 实例。

## 19.3 初始化 ManagerServlet

通常来说，servlet 对象由一个 org.apache.catalina.core.StandardWrapper 实例表示。在第一次调用 servlet 实例时，会先调用 StandardWrapper 对象的 loadServlet() 方法，然后调用 servlet 实例的 init() 方法（参见第 11 章内容）。而对于 ManagerServlet 类来说，在 loadServlet() 方法中有如下代码段：



```

...
// Special handling for ContainerServlet instances
if ((servlet instanceof ContainerServlet) &&
    isContainerProvidedServlet(actualClass)) {
    ((ContainerServlet). servlet).setWrapper(this);
}

// Call the initialization method of this servlet
try {
    instanceSupport.fireInstanceEvent (InstanceEvent.BEFORE_INIT_EVENT,
        servlet);
    servlet.init(facade);
}
...

```

其中，变量 `servlet` 表示将要载入的 `servlet` 类的实例。在这里，指的是 `ManagerServlet` 类。

在代码片段的 `if` 语句块中的代码表明，如果变量 `servlet` 指向的对象是 `org.apache.catalina.ContainerServlet` 类的实例，而且 `isContainerProvidedServlet()` 方法返回 `true`，就调用 `ContainerServlet` 接口的 `setWrapper()` 方法。

`ManagerServlet` 类实现 `ContainerServlet` 接口，因此，变量 `servlet` 引用的对象是 `ContainerServlet` 接口的实例。代码清单 19-2 给出了 `StandardWrapper` 类中 `isContainerProvidedServlet()` 方法的实现。

代码清单 19-2 `StandardWrapper` 类中 `isContainerProvidedServlet()` 方法的实现

```

private boolean isContainerProvidedServlet (String classname) {
    if (classname.startsWith("org.apache.catalina.")) {
        return (true);
    }
    try {
        Class clazz =
            this.getClass().getClassLoader().loadClass(classname);
        return (ContainerServlet.class.isAssignableFrom(clazz));
    }
    catch (Throwable t) {
        return (false);
    }
}

```

其中，传递给 `isContainerProvidedServlet()` 方法的参数 `classname` 是 `Managerservlet` 类的完全限定名，也就是 `org.apache.catalina.servlets.Managerservlet`。因此，`isContainerProvidedServlet()` 方法会返回 `true`。

如果参数 `classname` 指定的类是 `ContainerServlet` 接口的子类，即如果参数 `classname` 指定的是一个继承自 `ContainerServlet` 接口的接口，或是一个实现了 `ContainerServlet` 接口的类，则 `isContainerProvidedServlet()` 方法返回 `true`。

**注意** `java.lang.Class` 类的 `isAssignableFrom(Class clazz)` 方法会返回 `true`，如果当前 `Class` 对象所表示的类与参数 `clazz` 所表示的类与接口相同，或者是指定的 `clazz` 参数所表示的类与接口的父类和父接口。

因此，表示 `ManagerServlet` 类的 `StandardWrapper` 实例的 `loadservlet()` 方法会调用 `ManagerServlet` 类的 `setWrapper()` 方法。下面是 `ManagerServlet` 类的 `setWrapper()` 方法的实现：

```

public void setWrapper(Wrapper wrapper) {
    this.wrapper = wrapper;
    if (wrapper == null) {
        context = null;
        deployer = null;
    }
    else {
        context = (Context) wrapper.getParent();
        deployer = (Deployer) context.getParent();
    }
}

```

因为参数 wrapper 不为 null，所以会执 else 语句块行，其中的意思是变量 context 会被赋值为表示该 Manager 应用程序的 Context 实例，变量 deployer 被赋值为 Manager 应用程序的父容器，也就是 StandardHost 实例。其中，变量 deployer 尤为重要，因为它会在 ManagerServlet 类的好几个方法中调用。

在 StandardWrapper 类的 loadServlet() 方法调用了 ManagerServlet 类的 setWrapper() 方法后，loadServlet() 方法会调用 ManagerServlet 实例的 init() 方法。

## 19.4 列出已经部署的 Web 应用程序

可以通过访问如下 URL 来列出所有已经部署的 Web 应用程序：

`http://localhost:8080/manager/list`

下面是可能的输出结果：

```

OK - Listed applications for virtual host localhost
/admin:stopped:0:../server/webapps/admin
/appl:running:0:C:\123data\JavaProjects\Pymont\webapps\appl
/manager:running:0:../server/webapps/manager

```

上面的 URL 会调用 ManagerServlet 类的 list() 方法，该方法的实现代码在代码清单 19-3 中给出。

代码清单 19-3 ManagerServlet 类的 list() 方法的实现

```

protected void list(PrintWriter writer) {
    if (debug >= 1)
        log("list: Listing contexts for virtual host " +
            deployer.getName() + "");
    writer.println(sm.getString("managerServlet.listed",
        deployer.getName()));
    String contextPaths[] = deployer.findDeployedApps();
    for (int i = 0; i < contextPaths.length; i++) {
        Context context = deployer.findDeployedApp(contextPaths[i]);
        String displayPath = contextPaths[i];
        if (displayPath.equals(""))
            displayPath = "/";
        if (context != null) {
            if (context.getAvailable()) {
                writer.println(sm.getString("managerServlet.listitem",
                    displayPath, "running", ""
                        + context.getManager().findSessions().length,
                        context.getDocBase()));
            }
        }
    }
}

```

```

else {
    writer.println(sm.getString("managerServlet.listitem",
        displayPath, "stopped", "0", context.getDocBase()));
}
}
}
}

```

list() 方法会调用部署器的 findDeployedApps() 方法来获取 Catalina 中所有已经部署了的 Context 的路径。然后，对 path 数组进行迭代，获取每一个 Context，然后检查该 Context 是否可用。对每个可用的 Context，list() 方法输出上下文路径、字符串“running”、用户 session 的数量以及文档的根路径。对于那些不可用的 Context，list() 方法会输出上下文路径、字符串“stopped”、0 以及文档的根路径。

## 19.5 启动 Web 应用程序

可以使用如下的 URL 来启动某个 Web 应用程序：

```
http://localhost:8080/manager/start?path=/contextPath
```

其中，contextPath 是想要启动的 Web 应用程序的路径。例如，如果想要启动 admin 应用程序，可以使用如下的 URL：

```
http://localhost:8080/manager/start?path=/admin
```

若该 Web 应用程序已经启动，则返回错误通知。

在接收到该 URL 后，ManagerServlet 实例会调用代码清单 19-4 中给出的 start() 方法。

代码清单 19-4 ManagerServlet 类的 start() 方法的实现

```

protected void start(PrintWriter writer, String path) {
    if (debug >= 1)
        log("start: Starting web application at " + path + "");
    if ((path == null) || (!path.startsWith("/") && path.equals("")))) {
        writer.println(sm.getString("managerServlet.invalidPath", path));
        return;
    }
    String displayPath = path;
    if (path.equals("/") )
        path = "";
    try {
        Context context = deployer.findDeployedApp(path);
        if (context == null) {
            writer.println(sm.getString("managerServlet.noContext",
                displayPath));
            return;
        }
        deployer.start(path);
        if (context.getAvailable())
            writer.println
                (sm.getString("managerServlet.started", displayPath));
        else
            writer.println

```



```

        (sm.getString("managerServlet.startFailed", displayPath));
    }
    catch (Throwable t) {
        getServletContext().log
            (sm.getString("managerServlet.startFailed", displayPath), t);
        writer.println
            (sm.getString("managerServlet.startFailed", displayPath));
        writer.println(sm.getString("managerServlet.exception",
            t.toString()));
    }
}

```

在执行了一些检查工作后，start() 方法会在 try 代码块中调用部署器的 findDeployedApp() 方法。该方法返回参数 path 指向的 Context 对象。如果 context 不是 null，start() 方法调用部署器的 start() 方法来启动该 Web 应用程序。

## 19.6 关闭 Web 应用程序

可以使用如下的 URL 关闭某个 Web 应用程序：

http://localhost:8080/manager/stop?path=/contextPath

其中，contextPath 是希望关闭的 Web 应用程序的上下文路径。若该 Web 应用程序已经关闭，则返回一条错误消息。

当 ManagerServlet 实例接收到关闭请求后，它会调用 stop() 方法来完成关闭 Web 应用程序的工作。代码清单 19-5 给出了 stop() 方法的实现。

代码清单 19-5 ManagerServlet 类的 stop() 方法的实现

```

protected void stop(PrintWriter writer, String path) {
    if (debug >= 1)
        log("stop: Stopping web application at '" + path + "'");
    if ((path == null) || (!path.startsWith("/") && path.equals(""))) {
        writer.println(sm.getString("managerServlet.invalidPath", path));
        return;
    }
    String displayPath = path;
    if (path.equals("/") )
        path = "";

    try {
        Context context = deployer.findDeployedApp(path);
        if (context == null) {
            writer.println(sm.getString("managerServlet.noContext",
                displayPath));
            return;
        }
        // It isn't possible for the manager to stop itself
        if (context.getPath().equals(this.context.getPath())) {
            writer.println(sm.getString("managerServlet.noSelf"));
            return;
        }
        deployer.stop(path);
        writer.println(sm.getString("managerServlet.stopped",
            displayPath));
    }
}

```

```

catch (Throwable t) {
    log("ManagerServlet.stop[" + displayPath + "]", t);
    writer.println(sm.getString("managerServlet.exception",
        t.toString()));
}
}

```

你应该可以理解 stop() 方法是如何工作的。ManagerServlet 类中的其他方法也比较容易理解。

## 19.7 小结

在本章中，你学会了如何使用一个专用的接口 (ContainerServlet) 来创建一个可以访问 Catalina 内部类的 servlet 类。可以用来管理已部署 Web 应用程序的 Manager 应用程序展示了如何从 Wrapper 对象中获取其他 Catalina 对象。你可以设计一个用来管理 Tomcat 的、具有更高级功能的 servlet。

## 第 20 章

# 基于 JMX 的管理

第 19 章讨论了 Manager 应用程序。它展示了如何使用实现了 ContainerServlet 接口的 ManagerServlet 类来访问 Catalina 的内部对象。本章展示的对 Tomcat 的管理使用了更复杂一点儿的方法，使用 Java Management Extensions (JMX 规范) 来管理。如果你对 JMX 还不太熟悉，在本章的起始部分会对 JMX 做一个简单的介绍。此外，本章还会对 Commons Modeler 库进行介绍，Catalina 使用 Commons Modeler 库来简化编写托管 Bean 的工作。托管 Bean 就是用来管理 Catalina 中其他对象的 Bean。最后使用一个简单的示例程序来帮助你理解在 Tomcat 中 JMX 的使用方法。

### 20.1 JMX 简介

既然 ContainerServlet 接口已经有利用 Manager 应用程序访问 Catalina 的内部对象，那么为什么还要用 JMX 呢？因为 JMX 提供了比 ContainerServlet 接口更灵活的方法来管理 Tomcat。许多基于服务器的应用程序（如 Tomcat、JBoss、JONAS、Geronimo 等），都使用了 JMX 技术来管理各自的资源。

JMX 规范（本书中指 1.2.1 版本）定义了管理 Java 对象的公开标准。例如，Tomcat 4 和 Tomcat 5 使用 JMX 来启用 servlet 容器中的各种对象（如 Server 对象、Host 对象、Context 对象、Valve 对象等），这样更灵活，更易于管理应用程序管理。Tomcat 的开发者也编写了 Admin 应用程序来管理其他 Web 应用程序。

如果一个 Java 对象可以由一个遵循 JMX 规范的管理器应用程序管理，那么这个 Java 对象称为一个可由 JMX 管理的资源。实际上，一个可由 JMX 管理的资源可以是一个应用程序、一种实现、一个服务、一个设备、一个用户等。一个可由 JMX 管理的资源也是由 Java 编写，并提供了一个相应的 Java 包装。

若要使一个 Java 对象成为一个可由 JMX 管理的资源，则必须创建一个名为 Managed Bean 或 MBean 的对象。在 org.apache.catalina.mbeans 包下已经预定义了一些 MBean。在这个包中，ConnectorMBean、StandardEngineMBean、StandardHostMBean 和 StandardContextMBean 是 Managed Bean 的示例。从它们的名字中，你应该也可以猜到 ConnectMBean 用来管理连接器，StandardContextMBean 用来管理 org.apache.catalina.core.StandardContext 类的实例，以此类推。当然，你也可以自己编写一个 MBean 来管理多个 Java 对象。

MBean 会提供它所管理的一些 Java 对象的属性和方法供管理应用程序使用。管理应用程序本身并不能直接访问托管的 Java 对象。因此，可以选择 Java 对象的哪些属性和方法可以由管理应用程序使用。

当拥有了一个 MBean 类之后，需要将其实例化，并将其注册到另一个作为 MBean 服务器



的 Java 对象中。MBean 服务器中保存了应用程序中注册的所有的 MBean。管理应用程序通过 MBean 服务器来访问 MBean 实例。可以将 JMX 与 servlet 应用程序做个类比，管理应用程序的作用类似于 Web 浏览器，MBean 服务器的作用类似于 servlet 容器。MBean 服务器提供了使客户端，也就是管理应用程序，访问托管资源的方法。MBean 实例就好比是 servlet 类或 JSP 页面。Web 浏览器无法直接访问 servlet 实例或 JSP 页面，必须要通过 servlet 容器才可以；管理应用程序也必须要通过 MBean 服务器来访问 MBean 实例。

共有 4 种类型的 MBean，分别是标准类型、动态类型、开放类型和模型类型。其中标准类型的 MBean 最容易编写，但灵活性最低。其他 3 种类型的灵活型较好，其中我们对模型 MBean 尤其感兴趣，因为 Catalina 中就使用了这种类型的 MBean。标准 MBean 将在下一节介绍，并展示如何编写一个 MBean。然后，会对模型 MBean 进行讨论。我们会跳过动态 MBean 和开放 MBean，因为它们与本章内容无关。感兴趣的读者可以阅读 JMX 规范 1.2.1 来获取更多信息。

从结构上讲，JMX 规范分为 3 层：设备层、代理层和分布式服务层。MBean 服务器位于代理层，MBean 位于设备层。分布式服务层会在 JMX 规范将来的版本中涉及。

设备层规范定义了编写可由 JMX 管理的资源的标准，即如何编写 MBean。代理层定义了创建代理的规范。代理封装了 MBean 服务器，提供了处理 MBean 的服务。代理和它所管理的 MBean 通常都位于同一个 Java 虚拟机中。由于 JMX 规范附带了一个参考实现，因此并不需要自己编写 MBean 服务器。参考实现提供了创建默认 MBean 服务器的方法。

**注意** 可以从下面的地址中下载 JMX 规范和参考实现：<http://java.sun.com/products/JavaManagement/download.html>。MX4J 是一个开源版本的 JMX，其库可以在本书附带的软件中找到，也可以从下面的地址中下载：<http://mx4j.sourceforge.net>。

**警告** 本书附带的 ZIP 文件包含的 mx4j.jar 文件是 MX4J 的 2.0 beta 1 版本，替换了 Tomcat 4.1.12 中的 mx4j-jmx.jar 文件。这样做是为了可以使用更新版本的 JMX，也就是 1.2.1 版本的 JMX。

## 20.2 JMX API

JMX 的参考实现包含一个核心 Java 库，这个库位于 javax.management 包和 JMX 编程中具有特定功能的其他包下。本节将会讨论 JMX API 中一些比较重要的类。

### 20.2.1 MBeanServer 类

MBean 服务器是 javax.management.MBeanServer 接口的实例。要创建一个 MBeanServer 实例，只需要调用 javax.management.MBeanServerFactory 类的 createMBean() 方法即可。

要将一个 MBean 注册到 MBean 服务器中，可以调用 MBeanServer 实例的 registerMBean() 方法。下面是 registerMBean() 方法的签名：

```
public ObjectInstance registerMBean(java.lang.Object object,
    ObjectName name) throws InstanceAlreadyExistsException,
    MBeanRegistrationException, NotCompliantMBeanException
```

要调用 `registerMBean()` 方法, 需要传入一个待注册的 MBean 实例和一个 `ObjectName` 实例。`ObjectName` 实例与 `HashMap` 中的键类似, 它可以唯一地标识一个 MBean 实例。`registerMBean()` 方法会返回一个 `ObjectInstance` 实例。`javax.management.ObjectInstance` 类封装了一个 MBean 实例的对象名称和它的类名。

要想获取 MBean 实例或匹配某个模式的一组 MBean 实例, 可以使用 `MBeanServer` 接口提供的两个方法, 分别是 `queryNames()` 方法和 `queryMBeans()` 方法。`queryNames()` 方法返回一个 `java.util.Set` 实例, 其中包含了匹配某个指定模式对象名称的一组 MBean 实例的对象名称。下面是 `queryNames()` 方法的签名:

```
public java.util.Set queryNames(ObjectName name, QueryExp query)
```

其中, 参数 `query` 指定了过滤条件。

若参数 `name` 为 `null` 或者没有域, 而且指定了 `key` 属性, 那么会返回已经注册的 MBean 实例的所有 `ObjectName` 实例。如果参数 `query` 为 `null`, 则不会对查找对象进行过滤。

`queryMBeans()` 方法与 `queryNames()` 方法类似。它返回的也是一个 `java.util.Set` 实例, 但其中包含的是被选择的 MBeans 实例的 `ObjectInstance` 对象。`queryMBeans()` 方法的签名如下所示:

```
public java.util.Set queryMBeans(ObjectName name, QueryExp query)
```

一旦获得了所需要的 MBean 实例的对象名称, 就可以操作托管资源在 MBean 实例中提供的属性或调用其方法。

可以通过调用 `MBeanServer` 接口的 `invoke()` 方法调用已注册的 MBean 实例的任何方法。`MBeanServer` 接口的 `getAttribute()` 方法和 `setAttribute()` 方法用于获取或设置已注册的 MBean 实例的属性。

## 20.2.2 ObjectName 类

MBean 实例注册于 MBean 服务器中。MBean 服务器中的每个 MBean 实例都通过一个对象名称来唯一地标识, 就好像是 `HashMap` 中的每个条目都通过一个键来唯一地标识一样。

对象名称是 `javax.management.ObjectName` 类的实例。对象名称由两部分组成, 域和一个键/值对。域是一个字符串, 也可以是空字符串。在对象名称中, 域后接一个分号, 然后是一个或多个键/值对。在键/值对中, 键 (key) 是一个非空字符串, 并且不能包含下列字符: 等号、逗号、分号、星号和问号。在一个对象名称中, 同一个键只能出现一次。

键与其值是由等号分隔的, 键/值对之间由分号分隔。例如, 下面是一个有效的对象名称, 其中包含两个键:

```
myDomain:type=Car,color=blue
```

`ObjectName` 实例也表示在 MBean 服务器中搜索 MBean 实例的属性模式。`ObjectName` 实例可以在域部分或键/值对部分使用通配符来表示模式。作为模式的 `ObjectName` 实例可以有 0 个或多个键。

## 20.3 标准 MBean

标准 MBean 是最简单的 MBean 类型。要想通过标准 MBean 来管理一个 Java 对象，需要执行以下步骤：

- 创建一个接口，该接口的命名规范为：Java 类名 +MBean 后缀。例如，如果想要管理的 Java 类名为 Car，则需要创建的接口命名为 CarMBean；
- 修改 Java 类，让其实现刚刚创建的 CarMBean 接口；
- 创建一个代理，该代理类必须包含一个 MBeanServer 实例；
- 为 MBean 创建 ObjectName 实例；
- 实例化 MBeanServer 类；
- 将 MBean 注册到 MBeanServer 中。

标准 MBean 是最容易编写的 MBean 类型，但使用标准 MBean 就必须修改原有的 Java 类。在某些项目里，这不是问题，但是在其他一些项目（尤其是具有很多类的项目）中，这是不可接受的。其他类型的 MBean 允许在不修改原有 Java 类的基础上管理 Java 对象。

下面是一个标准 MBean 的例子，其中假设你想要使其成为 JMX 可管理的类是 Car：

```
package ex20.pyrmont.standardmbeantest;

public class Car {
    private String color = "red";

    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public void drive() {
        System.out.println("Baby you can drive my car.");
    }
}
```

第1步是修改 Car 类，使其实现 CarMBean 接口。代码清单 20-1 给出了新的 Car 类的定义。

代码清单 20-1 修改后的 Car 类

```
package ex20.pyrmont.standardmbeantest;

public class Car implements CarMBean {
    private String color = "red";

    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public void drive() {
        System.out.println("Baby you can drive my car.");
    }
}
```



现在要创建 CarMBean 接口。该接口的定义在代码清单 20-2 中给出。

代码清单 20-2 CarMBean 接口的定义

```
package ex20.pyrmont.standardmbeantest;

public interface CarMBean {
    public String getColor();
    public void setColor(String color);
    public void drive();
}
```

基本上来讲,要在接口中声明 Car 类中所要提供的所有方法。在这个例子中,在 CarMBean 接口中声明了 Car 类的所有方法。如果不希望 Car 类的 drive() 方法在管理应用程序中调用,只需要将 drive() 方法的定义从 CarMBean 接口中移除即可。

最后,代码清单 20-3 给出了用来创建标准 MBean 实例和管理 Car 对象的代理类 StandardAgent 的定义。

代码清单 20-3 StandardAgent 类的定义

```
package ex20.pyrmont.standardmbeantest;

import javax.management.Attribute;
import javax.management.ObjectName;
import javax.management.MBeanServer;
import javax.management.MBeanServerFactory;

public class StandardAgent {
    private MBeanServer mBeanServer = null;
    public StandardAgent() {
        mBeanServer = MBeanServerFactory.createMBeanServer();
    }
    public MBeanServer getMBeanServer() {
        return mBeanServer;
    }
    public ObjectName createObjectName(String name) {
        ObjectName objectName = null;
        try {
            objectName = new ObjectName(name);
        }
        catch (Exception e) {
        }
        return objectName;
    }
    private void createStandardBean(ObjectName objectName,
        String managedResourceClassName) {
        try {
            mBeanServer.createMBean(managedResourceClassName, objectName);
        }
        catch (Exception e) {
        }
    }

    public static void main(String[] args) {
        StandardAgent agent = new StandardAgent();
        MBeanServer mBeanServer = agent.getMBeanServer();
```

```

String domain = mBeanServer.getDefaultDomain();
String managedResourceClassName =
    "ex20.pyrmont.standardmbeantest.Car";
ObjectName objectName = agent.createObjectName(domain + ":type=" +
    managedResourceClassName);
agent.createStandardBean(objectName, managedResourceClassName);

// manage MBean
try {
    Attribute colorAttribute = new Attribute("Color", "blue");
    mBeanServer.setAttribute(objectName, colorAttribute);
    System.out.println(mBeanServer.getAttribute(objectName,
        "Color"));
    mBeanServer.invoke(objectName, "drive", null, null);
} catch (Exception e) {
    e.printStackTrace();
}
}

```

StandardAgent 类是一个代理类，用来实例化 MBean 服务器，并使用 MBean 服务器注册 CarMBean 实例。其中首先要注意的是变量 mBeanServer，StandardAgent 类的构造函数会将一个 MBeanServer 实例赋值给变量 mBeanServer。构造函数会调用 MBeanServerFactory 类的 createMBeanServer() 方法，创建一个 MBean 服务器实例：

```

public StandardAgent() {
    mBeanServer = MBeanServerFactory.createMBeanServer();
}

```

createMBeanServer() 方法会返回 JMX 参考实现的一个默认 MBean Server 对象。资深 JMX 程序员可能会实现自己的 MBeanServer，但这并不是本书所关心的重点。

代码清单 20-3 给出的 StandardAgent 类的 createObjectName() 方法会根据传入的字符串参数返回一个 ObjectName 实例。StandardAgent 类的 createStandardMBean() 方法会调用 MBeanServer 实例的 createMBean() 方法。createMBean() 方法接收托管资源的类名和一个 ObjectName 实例，该 ObjectName 实例唯一地标识了为托管资源创建的 MBean 实例。createMBean() 方法也会将创建的 MBean 实例注册到 MBeanServer 中。由于标准 MBean 实例遵循了特定的命名规则，因此不需要为 createMBean() 方法提供 MBean 的类名。如果托管资源的类名是 Car，则创建的 MBean 的类名是 CarMBean。

StandardAgent 类的主方法 main() 方法首先会创建 StandardAgent 类的一个实例，调用其 getMBeanServer() 方法，以得到 StandardAgent 中对 MBeanServer 实例的一次引用：

```

StandardAgent agent = new StandardAgent();
MBeanServer mBeanServer = agent.getMBeanServer();

```

然后，它将为 CarMBean 实例创建一个 ObjectName 实例。MBeanServer 实例的默认域会作为 ObjectName 实例的域使用。一个名为 type 的键会被添加到域后面。键 type 的值是托管资源的完全限定名：

```

String domain = mBeanServer.getDefaultDomain();
String managedResourceClassName =
    "ex20.pyrmont.standardmbeantest.Car";

```

```
ObjectName objectName = agent.createObjectName(domain + ":type=" +
    managedResourceClassName);
```

然后, `main()` 方法会调用 `createStandardBean()` 方法, 并传入对象名称和托管资源的类名:

```
agent.createStandardBean(objectName, managedResourceClassName);
```

接着, `main()` 方法就可以通过 `CarMBean` 实例来管理 `Car` 对象。它会创建一个名为 `colorAttribute` 的 `Attribute` 类型的对象, 用来表示 `Car` 类的 `Color` 属性, 并设置其值为 `blue`。然后, 它再调用 `setAttribute()` 方法, 并传入 `objectName` 实例和 `colorAttribute` 实例。接着, 它会通过调用 `MBeanServer` 对象的 `invoke()` 方法来调用 `Car` 实例的 `drive()` 方法:

```
// manage MBean
try {
    Attribute colorAttribute = new Attribute("Color", "blue");
    mBeanServer.setAttribute(objectName, colorAttribute);
    System.out.println(mBeanServer.getAttribute(objectName,
        "Color"));
    mBeanServer.invoke(objectName, "drive", null, null);
}
```

如果运行 `StandardAgent` 类, 可以在控制台中看到如下输出:

```
blue
Baby you can drive my car.
```

现在, 你可能很想知道, 我们到底为什么需要使用 JMX 来管理 Java 对象呢? 从上面的例子可以看到, 我们已经可以通过 `StandardAgent` 类来直接访问 `Car` 对象了。但是, 这里的关键问题是可以选择哪些功能需要暴露出来, 哪些方法需要对外隐藏。此外, 在 20.8 节中, 你会看到, `MBeanServer` 实例是作为托管对象和管理应用程序的中间层而存在的。

## 20.4 模型 MBean

相对于标准 MBean, 模型 MBean 更具灵活性。在编程上, 模型 MBean 难度更大一些, 但也不再需要为可管理的对象修改 Java 类了。如果不能修改已有的 Java 类, 那么使用模型 MBean 是不错的选择。

使用模型 MBean 与使用标准 MBean 有一些区别。在使用标准 MBean 管理资源时, 需要定义一个接口, 然后让托管资源实现该接口。而使用模型 MBean 时, 不需要定义接口。相反, 可以使用 `javax.management.modelmbean.ModelMBean` 接口来表示模型 MBean。只需要实现该接口。幸运的是, 在 JMX 的参考实现中有一个 `javax.management.modelmbean.RequiredModelMBean` 类, 是 `ModelMBean` 接口的默认实现。可以实例化 `RequiredModelMBean` 类或其子类。

**注意** 也可以使用 `ModelMBean` 接口的其他实现类。例如, 20.5 节将要介绍的 `Commons Modeler` 库有 `ModelMBean` 接口的一个自定义实现类, 后者并没有继承自 `RequiredModelMBean` 类。

编写一个模型 MBean 的最大挑战是告诉 `ModelMBean` 对象托管资源的哪些属性和方法可以暴露给代理。可以通过创建 `javax.management.modelmbean.ModelMBeanInfo` 对象来完成这个任务。`ModelMBeanInfo` 对象描述了将会暴露给代理的构造函数、属性、操作甚至是监听器。创建



ModelMBeanInfo 对象是一件枯燥的事情（参见本节中的例子），但当创建了该实例后，只需要将其与 ModelMBean 对象相关联即可。

使用 RequiredModelMBean 类作为 ModelMBean 的实现，有两种方法可以将 ModelMBean 对象与 ModelMBeanInfo 对象相关联：

- 1) 传入一个 ModelMBeanInfo 对象到 RequiredModelMBean 对象的构造函数中；
- 2) 调用 RequiredModelMBean 对象的 setModelMBeanInfo() 方法，并传入一个 ModelMBeanInfo 对象。

在创建了 ModelMBean 对象后，需要调用 ModelMBean 接口的 setManagedResource() 方法将其与托管资源相关联。该方法的签名如下所示：

```
public void setManagedResource(java.lang.Object managedResource,  
    java.lang.String managedResourceType) throws MBeanException,  
    RuntimeOperationsException, InstanceNotFoundException,  
    InvalidTargetObjectTypeException
```

参数 managedResourceType 的值可以是如下之一：ObjectReference、Handle、IOR、EJBHandle 或 RMIRReference。当前，只支持 ObjectReference。

当然，还需要创建一个 ObjectName 示例，并将 MBean 实例注册到 MBean 服务器中。

本节会提供一个标准的示例，使用模型 MBean 来管理 Car 对象。在讨论这个示例之前，需要先介绍一下 ModelMBeanInfo 接口。该接口的实例会将托管资源的属性和方法提供给代理层。

#### 20.4.1 MBeanInfo 接口与 ModelMBeanInfo 接口

javax.management.mbean.ModelMBeanInfo 接口描述了要通过 ModelMBean 暴露给代理层的构造函数、属性、方法和监听器。其中，构造函数是 javax.management.modelmbean.ModelMBeanConstructorInfo 类的实例，属性是 javax.management.modelmbean.ModelMBeanAttributeInfo 类的实例，方法是 javax.management.modelmbean.ModelMBeanOperationInfo 类的实例，监听器是 javax.management.modelmbean.ModelMBeanNotificationInfo 类的实例。在本章中，我们只关注属性和方法的使用。

JMX 提供了 ModelMBeanInfo 接口的默认实现，即 javax.management.modelmbean.ModelMBeanInfoSupport 类。下面是将在本节的示例中使用的 ModelMBeanInfoSupport 类的构造函数的签名：

```
public ModelMBeanInfoSupport(java.lang.String className,  
    java.lang.String description, ModelMBeanAttributeInfo[] attributes,  
    ModelMBeanConstructorInfo[] constructors,  
    ModelMBeanOperationInfo[] operations,  
    ModelMBeanNotificationInfo[] notifications)
```

可以通过调用 ModelMBeanAttributeInfo 类的构造函数来创建 ModelMBeanAttributeInfo 对象：

```
public ModelMBeanAttributeInfo(java.lang.String name,  
    java.lang.String type, java.lang.String description,  
    boolean isReadable, boolean isWritable,  
    boolean isIs, Descriptor descriptor)  
    throws RuntimeOperationsException
```

下面是参数列表:

- name, 属性的名称;
- type, 属性的类型名;
- description, 对属性的描述;
- isReadable, true 表示针对该属性有一个 getter 方法, false 表示没有;
- isWritable, true 表示针对该属性有一个 setter 方法, false 表示没有;
- isIs, true 表示针对该属性有一个 getter 方法, false 表示没有;
- descriptor, Descriptor 类的实例, 包含 Attribute 实例的适当元数据。如果它为 null, 会创建默认的 Descriptor 实例。

可以使用下面的构造函数创建一个 ModelMBeanOperationInfo 对象:

```
public ModelMBeanOperationInfo(java.lang.String name,
    java.lang.String description, MBeanParameterInfo[] signature,
    java.lang.String type, int impact, Descriptor)
    throws RuntimeException
```

下面是参数列表说明:

- name, 方法名;
- description, 方法描述;
- signature, MBeanParameterInfo 对象的数组, 描述了方法的参数;
- type, 方法返回值的类型;
- impact, 方法的影响, 可选值如下: INFO, ACTION, ACTION\_INFO 和 UNKNOWN;
- descriptor, Descriptor 实例, 包含 MBeanOperationInfo 实例的适当元数据。

#### 20.4.2 ModelMBean 示例

这个例子展示了如何使用模型 MBean 管理 Car 对象。代码清单 20-4 给出了 Car 类的定义。

代码清单 20-4 Car 类

```
package ex20.pyrmont.modelmbeantest1;

public class Car {
    private String color = "red";
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public void drive() {
        System.out.println("Baby you can drive my car.");
    }
}
```

对于模型 MBean, 不需要向使用标准 MBean 那样, 编写一个接口。只需要实例化 RequiredMBean 类。代码清单 20-5 给出了 ModelAgent 类的定义。该类用来创建模型 MBean 的实例, 并管理 Car 对象。

代码清单 20-5 ModelAgent 类

```

package ex20.pyrmont.modelmbeantest1;

import javax.management.Attribute;
import javax.management.Descriptor;
import javax.management.MalformedObjectNameException;
import javax.management.MBeanOperationInfo;
import javax.management.MBeanParameterInfo;
import javax.management.MBeanServer;
import javax.management.MBeanServerFactory;
import javax.management.ObjectName;
import javax.management.modelmbean.DescriptorSupport;
import javax.management.modelmbean.ModelMBean;
import javax.management.modelmbean.ModelMBeanAttributeInfo;
import javax.management.modelmbean.ModelMBeanInfo;
import javax.management.modelmbean.ModelMBeanInfoSupport;
import javax.management.modelmbean.ModelMBeanOperationInfo;
import javax.management.modelmbean.RequiredModelMBean;

public class ModelAgent {

    private String MANAGED_CLASS_NAME =
        "ex20.pyrmont.modelmbeantest1.Car";
    private MBeanServer mBeanServer = null;

    public ModelAgent() {
        mBeanServer = MBeanServerFactory.createMBeanServer();
    }

    public MBeanServer getMBeanServer() {
        return mBeanServer;
    }

    private ObjectName createObjectName(String name) {
        ObjectName objectName = null;
        try {
            objectName = new ObjectName(name);
        } catch (MalformedObjectNameException e) {
            e.printStackTrace();
        }
        return objectName;
    }

    private ModelMBean createMBean(ObjectName objectName,
        String mbeanName) {
        ModelMBeanInfo mBeanInfo = createModelMBeanInfo(objectName,
            mbeanName);
        RequiredModelMBean modelMBean = null;
        try {
            modelMBean = new RequiredModelMBean(mBeanInfo);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return modelMBean;
    }

    private ModelMBeanInfo createModelMBeanInfo(ObjectName
        inMbeanObjectName, String inMbeanName) {
        ModelMBeanInfo mBeanInfo = null;
        ModelMBeanAttributeInfo[] attributes = new

```



```

ModelMBeanAttributeInfo[1];
ModelMBeanOperationInfo[] operations = new
ModelMBeanOperationInfo[3];
try {
    attributes[0] = new ModelMBeanAttributeInfo("Color",
        "java.lang.String",
        "the color.", true, true, false, null);
    operations[0] = new ModelMBeanOperationInfo("drive",
        "the drive method",
        null, "void", MBeanOperationInfo.ACTION, null);
    operations[1] = new ModelMBeanOperationInfo("getColor",
        "get color attribute",
        null, "java.lang.String", MBeanOperationInfo.ACTION, null);

    Descriptor setColorDesc = new DescriptorSupport(new String[] {
        "name=setColor", "descriptorType=operation",
        "class=" + MANAGED_CLASS_NAME, "role=operation"});
    MBeanParameterInfo[] setColorParams = new MBeanParameterInfo[] {
        (new MBeanParameterInfo("new color", "java.lang.String",
            "new Color value")) };
    operations[2] = new ModelMBeanOperationInfo("setColor",
        "set Color attribute", setColorParams, "void",
        MBeanOperationInfo.ACTION, setColorDesc);

    mBeanInfo = new ModelMBeanInfoSupport(MANAGED_CLASS_NAME,
        null, attributes, null, operations, null);
} catch (Exception e) {
    e.printStackTrace();
}
return mBeanInfo;
}

public static void main(String[] args) {
    ModelAgent agent = new ModelAgent();
    MBeanServer mBeanServer = agent.getMBeanServer();
    Car car = new Car();
    String domain = mBeanServer.getDefaultDomain();
    ObjectName objectName = agent.createObjectName(domain +
        ":type=MyCar");
    String mBeanName = "myMBean";
    ModelMBean modelMBean = agent.createMBean(objectName, mBeanName);
    try {
        modelMBean.setManagedResource(car, "ObjectReference");
        mBeanServer.registerMBean(modelMBean, objectName);
    } catch (Exception e) {
    }

    // manage the bean
    try {
        Attribute attribute = new Attribute("Color", "green");
        mBeanServer.setAttribute(objectName, attribute);

        String color = (String) mBeanServer.getAttribute(objectName,
            "Color");
        System.out.println("Color:" + color);

        attribute = new Attribute("Color", "blue");
        mBeanServer.setAttribute(objectName, attribute);
        color = (String) mBeanServer.getAttribute(objectName, "Color");
        System.out.println("Color:" + color);
    }
}

```

```
        mBeanServer.invoke(objectName, "drive", null, null);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

如你所见，编写一个模型 MBean 需要做很多工作，尤其是声明托管资源所要暴露的所有属性和方法。20.5 节将会介绍 Commons Modeler 库。使用 Commons Modeler 库有助于编写模型 MBean。

## 20.5 Commons Modeler 库

Commons Modeler 库是 Apache 软件基金会的 Jakarta 项目的一部分，目的是使编写模型 MBean 更加方便。实际上，最大的帮助是不需要再写代码创建 ModelMBeanInfo 对象了。

回忆前面的章节的例子，在创建 RequiredModelMBean 实例时，需要创建一个 ModelMBeanInfo 对象，并将其传给 RequiredModelMBean 类的构造函数：

```
ModelMBeanInfo mBeanInfo = createModelMBeanInfo(objectName,
    mbeanName);
RequiredModelMBean modelMBean = null;
try {
    modelMBean = new RequiredModelMBean(mBeanInfo);
}
...
```

ModelMBeanInfo 对象描述了将要由 MBean 实例暴露出的属性和方法。实现 createModelMBeanInfo() 方法是枯燥的，因为必须列出所有要暴露的属性和方法并将它们传给 ModelMBeanInfo 对象。

而使用 Commons Modeler 库，就不再需要创建 ModelMBeanInfo 对象。相反，对模型 MBean 的描述被封装在一个 org.apache.catalina.modeler.ManagedBean 对象中。不需要编写代码在 MBean 中暴露出属性和方法。只需要编写一个 mbean 的描述符文件（一个 XML 文档），列出想要创建的 MBean。对于每个 MBean，需要写出 MBean 类和托管资源类的完全限定名，此外还有由 MBean 暴露出的属性和方法。然后，使用 org.apache.commons.modeler.Registry 实例读取这个 XML 文档，并创建一个 MBeanServer 实例，在按照 mbean 描述符文件中的 XML 元素创建所有的 ManagedBean 实例。

然后，调用 ManagedBean 实例的 createMBean() 方法创建模型 MBean。这之后就是普通的流程了。需要创建 ObjectName 实例，并将其与 MBean 实例一起注册到 MBean 服务器中。下面会先介绍一下 mbean 描述符文件的格式，然后讨论 Modeler 库中的 3 个重要的类，分别是 Registry 类、ManagedBean 类和 BaseModelMBean 类。

**注意** Tomcat 4 使用的仍然是老版本的 Modeler 库，其中有一些当前已经废弃的方法。在本章中，仍然会使用 Tomcat 4 中的 Modeler 库，目的是使你更好地了解 org.apache.catalina.mbeans 包中与 MBean 相关的类的工作原理。

### 20.5.1 MBean 描述符

MBean 描述符是一个 XML 文档, 该文档描述了将会由 MBean 服务器管理的模型 MBean 的实例。MBean 描述符以下面的头信息开始:

```
<?xml version="1.0"?>
<!DOCTYPE mbeans-descriptors PUBLIC
"-//Apache Software Foundation//DTD Model MBeans Configuration File"
"http://jakarta.apache.org/commons/dtds/mbeans-descriptors.dtd">
```

接下来是 mbeans-descriptors 的根元素:

```
<mbeans-descriptors>
...
</mbeans-descriptors>
```

在开始和末尾的 mbeans-descriptors 标签内部是 mbean 元素, 每个 mbean 标签表示一个模型 MBean。mbean 元素包含了分别用来表示属性、方法、构造函数和通知的元素。下面几节会对此进行分别讨论, 从而有助于你理解 Tomcat 的 MBean 描述符。

#### 1. mbean 元素

mbean 元素描述一个模型 MBean, 包含创建对应 ModelMBeanInfo 对象的信息。mbean 元素的定义如下所示:

```
<!ELEMENT mbean (descriptor?, attribute*, constructor*, notification*,
operation*)>
```

mbean 元素定义了具体的规范, mbean 元素可以有一个可选的 descriptor 元素, 0 个或多个 attribute 元素, 0 个或多个 constructor 元素, 0 个或多个 notification 元素, 0 个或多个 operation 元素。

mbean 元素可以有如下的属性:

- className, 实现 ModelMBean 接口的 Java 类的完全限定名, 若该属性未赋值, 则默认使用 org.apache.commons.modeler.BaseModelMBean 类;
- description, 该模型 MBean 的简单描述;
- domain, 在创建 ModelMBean 的 ObjectName 时, 托管的 bean 创建的 ModelMBean 实例被注册到的 MBean 服务器的域名;
- group, 组分类的可选名, 可以用来选择具有相似 MBean 实现类的组;
- name, 唯一标识模型 MBean 的名称, 一般情况下, 会使用相关服务器组件的基类名;
- type, 托管资源实现类的完全限定的 Java 类名。

#### 2. attribute 元素

使用 attribute 元素描述 MBean 的 JavaBean 属性。attribute 元素有一个可选的 descriptor 元素, attribute 元素有如下可选属性:

- description, 该属性的简单描述;
- displayName, 该属性的显示名称;
- getMethod, 由 attribute 元素表示的属性的 getter 方法;
- is, 一个布尔值, 指明该属性是否是一个布尔值, 是否有 getter 方法。默认情况下, 该属



性值为 false;

- name, 该 JavaBean 属性的名称;
- readable, 一个布尔值, 指明该属性对管理应用程序来说是否可读, 该属性值默认为 true;
- setMethod, 由 attribute 元素表示的属性的 setter 方法;
- type, 该属性的完全限定的 Java 类名;
- writeable, 一个布尔值, 表明该属性对管理应用程序来说是否可写, 该属性值默认为 true。

### 3. operation 元素

operation 元素描述了模型 MBean 中要暴露给管理应用程序的公共方法, 它可以有 0 个或多个 parameter 子元素和如下的属性:

- description, 方法的简单描述;
- impact, 指明方法的影响, 可选值为, ACTION、ACTION-INFO、INFO 或 UNKNOWN;
- name, 公共方法的名称;
- returnType, 方法返回值的完全限定的 Java 类名。

### 4. parameter 元素

parameter 元素描述了将要传递给构造函数或方法的参数。它可以有如下的属性:

- description, 该参数的简单描述;
- name, 参数名;
- type, 参数的完全限定的 Java 类名。

## 20.5.2 mbean 元素示例

在 Catalina 的 mbean-descriptors.xml 文件中声明了一系列模型 MBean, 该文件位于 org.apache.catalina.mbeans 包下。代码清单 20-6 给出了 mbean-descriptors.xml 文件中对 StandardServer MBean 的声明。

代码清单 20-6 StandardServer MBean 的声明

```
<mbean name="StandardServer"
  className="org.apache.catalina.mbeans.StandardServerMBean"
  description="Standard Server Component"
  domain="Catalina"
  group="Server"
  type="org.apache.catalina.core.StandardServer">

  <attribute name="debug"
    description="The debugging detail level for this component"
    type="int"/>
  <attribute name="managedResource"
    description="The managed resource this MBean is associated with"
    type="java.lang.Object"/>
  <attribute name="port"
    description="TCP port for shutdown messages"
    type="int"/>
  <attribute name="shutdown"
    description="Shutdown password"
```

```
type="java.lang.String"/>
<operation name="store">
  description="Save current state to server.xml file"
  impact="ACTION"
  returnType="void">
</operation>
</mbean>
```

代码清单 20-6 中的 mbean 元素声明了一个模型 MBean，其唯一标识是 StandardServer。该 MBean 是 org.apache.catalina.mbeans.StandardServerMBean 类的一个对象，负责管理 org.apache.catalina.core.StandardServer 类的对象。domain 属性的值是 Catalina，group 属性的值是 Server。

模型 MBean 暴露出的属性有 4 个，分别是 debug、managedResource、port 和 shutdown，正如 mbean 元素中嵌套的 4 个 attribute 元素所描述的一样。此外，MBean 还暴露出了一个方法，即 store() 方法，由 operation 元素描述。

### 20.5.3 自己编写一个模型 MBean 类

使用 Commons Modeler 库，需要在 mbean 元素的 className 属性中指明自定义的模型 MBean 的类型。默认情况下，Commons Modeler 库使用 org.apache.commons.modeler.BaseModelMBean 类。有以下两种情况，其中可能需要对 BaseModelMBean 类进行扩展：

- 需要覆盖托管资源的属性或方法；
- 需要添加在托管资源中没有定义的属性或方法。

在 org.apache.catalina.mbeans 包下，Catalina 提供了 BaseModelMBean 类的很多子类，可以用来实现上述需求。

### 20.5.4 Registry 类

org.apache.commons.modeler.Registry 类定义了很多方法，下面是可以使用该类做的事情：

- 获取 javax.management.MBeanServer 类的一个实例，所以不再需要调用 javax.management.MBeanServerFactory 类的 createMBeanServer() 方法了；
- 使用 loadRegistry() 方法读取 MBean 的描述符文件；
- 创建一个 ManagedBean 对象，用于创建模型 MBean 的实例。

### 20.5.5 ManagedBean

ManagedBean 对象描述了一个模型 MBean，该类用于取代 javax.management.MBeanInfo 对象。

### 20.5.6 BaseModelMBean

org.apache.commons.modeler.BaseModelMBean 类实现了 javax.management.modelmbean.ModelMBean 接口。使用这个类，就不需要用 javax.management.modelmbean.RequiredModelMBean 类了。

该类用一个比较有用的字段是 resource 字段。resource 字段表示该模型 MBean 管理的资源。resource 字段的定义如下所示：

```
protected java.lang.Object resource;
```

## 20.5.7 使用 Modeler 库 API

代码清单 20-7 给出了想要管理其对象的 Car 类的定义。

代码清单 20-7 Car 类

```
package ex20.pyrmont.modelmbeantest2;

public class Car {
    public Car() {
        System.out.println("Car constructor");
    }
    private String color = "red";

    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public void drive() {
        System.out.println("Baby you can drive my car.");
    }
}
```

使用 Commons Modeler 库，不需要使用硬编码的方式，将托管对象的所有属性和方法都写在代码中。相反，可以将它们写在一个 XML 文档中，作为 MBean 的描述符文件。在这个例子中，这样的文档是 car-mbean-descriptor.xml 文件，具体内容在代码清单 20-8 中给出。

代码清单 20-8 car-mbean-descriptor.xml 文件

```
<?xml version="1.0"?>
<!DOCTYPE mbeans-descriptors PUBLIC
"-//Apache Software Foundation//DTD Model MBeans Configuration File"
"http://jakarta.apache.org/commons/dtds/mbeans-descriptors.dtd">

<mbeans-descriptors>
  <mbean name="myMBean"
    className="javax.management.modelmbean.RequiredModelMBean"
    description="The ModelMBean that manages our Car object"
    type="ex20.pyrmont.modelmbeantest.Car">

    <attribute name="Color"
      description="The car color"
      type="java.lang.String"/>
    <operation name="drive"
      description="drive method"
      impact="ACTION"
      returnType="void">
      <parameter name="driver" description="the driver parameter"
        type="java.lang.String"/>
    </operation>
  </mbean>
</mbeans-descriptors>
```



现在，需要一个代理类 ModelAgent.java，其具体定义见代码清单 20-9。

代码清单 20-9 ModelAgent 类

```
package ex20.pyrmont.modelmbeanTest2;

import java.io.InputStream;
import java.net.URL;
import javax.management.Attribute;
import javax.management.MalformedObjectNameException;
import javax.management.MBeanServer;
import javax.management.ObjectName;
import javax.management.modelmbean.ModelMBean;

import org.apache.commons.modeler.ManagedBean;
import org.apache.commons.modeler.Registry;

public class ModelAgent {
    private Registry registry;
    private MBeanServer mBeanServer;

    public ModelAgent() {
        registry = createRegistry();
        try {
            mBeanServer = Registry.getServer();
        }
        catch (Throwable t) {
            t.printStackTrace(System.out);
            System.exit(1);
        }
    }

    public MBeanServer getMBeanServer() {
        return mBeanServer;
    }

    public Registry createRegistry() {
        Registry registry = null;
        try {
            URL url = ModelAgent.class.getResource
                ("/ex20/pyrmont/modelmbeanTest2/car-mbean-descriptor.xml");
            InputStream stream = url.openStream();
            Registry.loadRegistry(stream);
            stream.close();
            registry = Registry.getRegistry();
        }
        catch (Throwable t) {
            System.out.println(t.toString());
        }
        return (registry);
    }

    public ModelMBean createModelMBean(String mBeanName)
        throws Exception {
        ManagedBean managed = registry.findManagedBean(mBeanName);
        if (managed == null) {
            System.out.println("ManagedBean null");
            return null;
        }
        ModelMBean mbean = managed.createMBean();
        ObjectName objectName = createObjectName();
        return mbean;
    }
}
```

```

}

private ObjectName createObjectName() {
    ObjectName objectName = null;
    String domain = mBeanServer.getDefaultDomain();
    try {
        objectName = new ObjectName(domain + ":type=MyCar");
    }
    catch (MalformedObjectNameException e) {
        e.printStackTrace();
    }
    return objectName;
}

public static void main(String[] args) {
    ModelAgent agent = new ModelAgent();
    MBeanServer mBeanServer = agent.getMBeanServer();
    Car car = new Car();
    System.out.println("Creating ObjectName");
    ObjectName objectName = agent.createObjectName();
    try {
        ModelMBean modelMBean = agent.createModelMBean("myMBean");
        modelMBean.setManagedResource(car, "ObjectReference");
        mBeanServer.registerMBean(modelMBean, objectName);
    }
    catch (Exception e) {
        System.out.println(e.toString());
    }
    // manage the bean
    try {
        Attribute attribute = new Attribute("Color", "green");
        mBeanServer.setAttribute(objectName, attribute);
        String color = (String) mBeanServer.getAttribute(objectName,
            "Color");
        System.out.println("Color:" + color);

        attribute = new Attribute("Color", "blue");
        mBeanServer.setAttribute(objectName, attribute);
        color = (String) mBeanServer.getAttribute(objectName, "Color");
        System.out.println("Color:" + color);
        mBeanServer.invoke(objectName, "drive", null, null);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}

```

使用 Commons Modeler 库后，是不是减少了很多代码呢？

## 20.6 Catalina 中的 MBean

正如本章开头所说，Catalina 在 `org.apache.catalina.mbeans` 包中提供了一系列 MBean 类。这些 MBean 类都直接或间接继承自 `org.apache.commons.modeler.BaseModelMBean` 类。本节讨论 Tomcat 4 中 3 个最重要的 MBean，分别是 `ClassNameMBean` 类、`StandardServerMBean` 类和 `MBeanFactory` 类。如果你能够理解这 3 个类的工作机制，那么对于 Catalina 中的其他模型

MBean 类也就都能够理解了。本节将会对这 3 个类以及 org.apache.catalina.mbeans 包下的一个工具类 MBeanUtil 进行讲解。

### 20.6.1 ClassNameMBean 类

org.apache.catalina.mbeans.ClassNameMBean 类继承自 org.apache.commons.modeler.BaseModelMBean 类。它提供了一个只写属性 className，用于表示托管资源的类名。代码清单 20-10 给出了 ClassNameMBean 类的定义。

代码清单 20-10 ClassNameMBean 类

```
package org.apache.catalina.mbeans;

import javax.management.MBeanException;
import javax.management.RuntimeOperationsException;
import org.apache.commons.modeler.BaseModelMBean;

public class ClassNameMBean extends BaseModelMBean {
    public ClassNameMBean()
        throws MBeanException, RuntimeOperationsException {
        super();
    }
    public String getClassName() {
        return (this.resource.getClass().getName());
    }
}
```

ClassNameMBean 类是 BaseModelMBean 的子类，其只写属性 className 在托管资源中不可见。mbeans-descriptors.xml 文件中的很多 mbean 元素使用该类作为其模型 MBean 的类型。

### 20.6.2 StandardServerMBean 类

StandardServerMBean 类继承自 org.apache.commons.modeler.BaseModelMBean 类，用于管理 org.apache.catalina.core.StandardServer 类的实例。StandardServerMBean 类是模型 MBean 类的一个示例，它重写了托管资源的 store() 方法。当管理应用程序调用 store() 方法时，实际上会执行 StandardServerMBean 实例的 store() 方法，而不是托管的 StandardServer 对象的 store() 方法。代码清单 20-11 给出了 StandardServerMBean 类的定义。

代码清单 20-11 StandardServerMBean 类

```
package org.apache.catalina.mbeans;

import javax.management.InstanceNotFoundException;
import javax.management.MBeanException;
import javax.management.MBeanServer;
import javax.management.RuntimeOperationsException;
import org.apache.catalina.Server;
import org.apache.catalina.ServerFactory;
import org.apache.catalina.core.StandardServer;
import org.apache.commons.modeler.BaseModelMBean;

public class StandardServerMBean extends BaseModelMBean {
```



```

private static MBeanServer mserver = MBeanUtils.createServer();
public StandardServerMBean()
    throws MBeanException, RuntimeOperationsException {
    super();
}

public synchronized void store() throws InstanceNotFoundException,
    MBeanException, RuntimeOperationsException {

    Server server = ServerFactory.getServer();
    if (server instanceof StandardServer) {
        try {
            ((StandardServer) server).store();
        }
        catch (Exception e) {
            throw new MBeanException(e, "Error updating conf/server.xml");
        }
    }
}
}

```

StandardServerMBean 类是一种模型 MBean，继承自 BaseModelMBean 类，并重写了托管资源的一个方法。

### 20.6.3 MBeanFactory 类

MBeanFactory 类的实例是一个工厂对象，用于创建管理 Tomcat 中各种资源的所有模型 Mbean。MBeanFactory 类还提供了删除这些 MBean 的方法。

例如，代码清单 20-12 给出了 MBeanFactory 类的 createStandardContext() 方法的实现。

代码清单 20-12 createStandardContext() 方法的实现

```

public String createStandardContext(String parent,
    String path, String docBase) throws Exception {
    // Create a new StandardContext instance
    StandardContext context = new StandardContext();
    path = getPathStr(path);
    context.setPath(path);
    context.setDocBase(docBase);
    ContextConfig contextConfig = new ContextConfig();
    context.addLifecycleListener(contextConfig);

    // Add the new instance to its parent component
    ObjectName pname = new ObjectName(parent);
    Server server = ServerFactory.getServer();
    Service service =
        server.findService(pname.getKeyProperty("service"));
    Engine engine = (Engine) service.getContainer();
    Host host = (Host) engine.findChild(pname.getKeyProperty("host"));

    // Add context to the host
    host.addChild(context);

    // Return the corresponding MBean name
    ManagedBean managed = registry.findManagedBean("StandardContext");
    ObjectName oname =

```

```

MBeanUtils.createObjectName(managed.getDomain(), context);
return (oname.toString());
}

```

## 20.6.4 MBeanUtil

org.apache.catalina.mbeans.MBeanUtil 类是一个工具类，提供了一些静态方法用于创建各种管理 Catalina 对象的 MBean，删除 MBean，以及创建 ObjectName 实例等。例如，代码清单 20-13 给出的 createMBean() 方法用于创建一个管理 org.apache.catalina.Server 对象的模型 MBean。

代码清单 20-13 createMBean() 方法的定义

```

public static ModelMBean createMBean(Server server) throws Exception {
    String mname = createManagedName(server);
    ManagedBean managed = registry.findManagedBean(mname);
    if (managed == null) {
        Exception e = new Exception(
            "ManagedBean is not found with "+mname);
        throw new MBeanException(e);
    }
    String domain = managed.getDomain();
    if (domain == null)
        domain = mserver.getDefaultDomain();
    ModelMBean mbean = managed.createMBean(server);
    ObjectName oname = createObjectName(domain, server);
    mserver.registerMBean(mbean, oname);
    return (mbean);
}

```

## 20.7 创建 Catalina 的 MBean

现在，你已经熟悉了 Catalina 中的一些模型 MBean，下面要介绍一下如何创建这些 MBean，并使用它们来管理 Catalina 中的托管资源。

在 Tomcat 的配置文件 server.xml 中，在 Server 元素里面定义了如下所示的 Listener 元素：

```

<Server port="8005" shutdown="SHUTDOWN" debug="0">
  <Listener
    className="org.apache.catalina.mbeans.ServerLifecycleListener"
    debug="0"/>
  ...

```

这里为服务器组件 org.apache.catalina.core.StandardServer 类的实例添加了一个 org.apache.catalina.mbeans.ServerLifecycleListener 类型的监听器。当 StandardServer 实例启动时，它会触发 START\_EVENT 事件，如 StandardServer 类中定义的 start() 方法所示：

```

public void start() throws LifecycleException {
    ...
    lifecycle.fireLifecycleEvent(START_EVENT, null);
    ...
}

```

而当 StandardServer 对象关闭时，会触发 STOP\_EVENT 事件，如 StandardServer 类中定义的 stop() 方法所示：

```
public void stop() throws LifecycleException {
    ...
    lifecycle.fireLifecycleEvent(STOP_EVENT, null);
    ...
}
```

这些事件会执行 ServerLifecycleListener 类的 lifecycleEvent() 方法。代码清单 20-14 给出了 lifecycleEvent() 方法的实现。

代码清单 20-14 ServerLifecycleListener 类中 lifecycleEvent() 方法的实现

```
public void lifecycleEvent(LifecycleEvent event) {
    Lifecycle lifecycle = event.getLifecycle();
    if (Lifecycle.START_EVENT.equals(event.getType())) {
        if (lifecycle instanceof Server) {
            // Loading additional MBean descriptors
            loadMBeanDescriptors();
            createMBeans();
        }
    }
    else if (Lifecycle.STOP_EVENT.equals(event.getType())) {
        if (lifecycle instanceof Server) {
            destroyMBeans();
        }
    }
    else if (Context.RELOAD_EVENT.equals(event.getType())) {
        if (lifecycle instanceof StandardContext) {
            StandardContext context = (StandardContext)lifecycle;
            if (context.getPrivileged()) {
                context.getServletContext().setAttribute
                    (Globals.MBEAN_REGISTRY_ATTR,
                     MBeanUtils.createRegistry());
                context.getServletContext().setAttribute
                    (Globals.MBEAN_SERVER_ATTR,
                     MBeanUtils.createServer());
            }
        }
    }
}
```

在 Catalina 中，createMBeans() 方法用来创建所有的 MBean 实例。该方法首先会创建 MBeanFactory 类的一个实例，也是一个模型 MBean 实例，如代码清单 20-15 所示。

代码清单 20-15 ServerLifecycleListener 类的 createMBeans() 方法的实现

```
protected void createMBeans() {
    try {
        MBeanFactory factory = new MBeanFactory();
        createMBeans(factory);
        createMBeans(ServerFactory.getServer());
    }
    catch (MBeanException t) {
        Exception e = t.getTargetException();
        if (e == null)
            e = t;
    }
}
```



```
        log("createMBeans: MBeanException", e);
    }
    catch (Throwable t) {
        log("createMBeans: Throwable", t);
    }
}
```

第一个 createMBeans() 方法会使用 MBeanUtil 类为 MBeanFactory 实例创建一个 ObjectName, 然后将其注册到 MBean 服务器中。

第二个 createMBeans() 方法接受一个 org.apache.catalina.Server 对象, 为其创建一个模型 MBean。代码清单 20-16 给出了 createMBeans() 方法创建模型 MBean 的具体过程。

代码清单 20-16 createMBeans() 方法创建 Server 对象的 MBean 实例的过程

```
protected void createMBeans(Server server) throws Exception {
    // Create the MBean for the Server itself
    if (debug >= 2)
        log("Creating MBean for Server " + server);
    MBeanUtils.createMBean(server);
    if (server instanceof StandardServer) {
        ((StandardServer) server).addPropertyChangeListener(this);
    }

    // Create the MBeans for the global NamingResources (if any)
    NamingResources resources = server.getGlobalNamingResources();
    if (resources != null) {
        createMBeans(resources);
    }

    // Create the MBeans for each child Service
    Service services[] = server.findServices();
    for (int i = 0; i < services.length; i++) {
        // FIXME - Warp object hierarchy not currently supported
        if (services[i].getContainer().getClass().getName().equals(
            "org.apache.catalina.connector.warp.WarpEngine")) {
            if (debug >= 1) {
                log("Skipping MBean for Service " + services[i]);
            }
            continue;
        }
        createMBeans(services[i]);
    }
}
```

注意, 代码清单 20-16 中的 createMBeans() 方法调用 for 循环中下面的语句遍历 StandardServer 实例中所有的 Service 对象:

```
createMBeans(services[i]);
```

该方法为每个 Service 对象创建 MBean, 然后, 为每个 Service 对象中所有的连接器和 Engine 对象调用 createMBeans() 方法创建 MBean 实例。代码清单 20-17 给出了为 Service 实例创建 MBean 的 createMBeans() 方法的实现。

代码清单 20-17 为 Service 实例创建 MBean 的 createMBeans() 方法的实现

```
protected void createMBeans(Service service) throws Exception {
    // Create the MBean for the Service itself
    if (debug >= 2)
        log("Creating MBean for Service " + service);
    MBeanUtils.createMBean(service);
    if (service instanceof StandardService) {
        ((StandardService) service).addPropertyChangeListener(this);
    }

    // Create the MBeans for the corresponding Connectors
    Connector connectors[] = service.findConnectors();
    for (int j = 0; j < connectors.length; j++) {
        createMBeans(connectors[j]);
    }

    // Create the MBean for the associated Engine and friends
    Engine engine = (Engine) service.getContainer();
    if (engine != null) {
        createMBeans(engine);
    }
}
```

正如你想的那样，语句 `createMBeans(engine)` 会调用 `createMBeans()` 方法为每个 Host 实例创建 MBean：

```
protected void createMBeans(Engine engine) throws Exception {
    // Create the MBean for the Engine itself
    if (debug >= 2) {
        log("Creating MBean for Engine " + engine);
    }
    MBeanUtils.createMBean(engine);
    ...
    Container hosts[] = engine.findChildren();
    for (int j = 0; j < hosts.length; j++) {
        createMBeans((Host) hosts[j]);
    }
    ...
}
```

接着，`createMBeans(host)` 方法会使用 `createMBeans()` 方法，为每个 Context 实例创建 MBean：

```
protected void createMBeans(Host host) throws Exception {
    ...
    MBeanUtils.createMBean(host);
    ...
    Container contexts[] = host.findChildren();
    for (int k = 0; k < contexts.length; k++) {
        createMBeans((Context) contexts[k]);
    }
    ...
}
```

`createMBeans(context)` 方法的实现如下所示：

```
protected void createMBeans(Context context) throws Exception {
    ...
    MBeanUtils.createMBean(context);
    ...
    context.addContainerListener(this);
}
```

```

if (context instanceof StandardContext) {
    ((StandardContext) context).addPropertyChangeListener(this);
    ((StandardContext) context).addLifecycleListener(this);
}

// If the context is privileged, give a reference to it
// in a servlet context attribute
if (context.getPrivileged()) {
    context.getServletContext().setAttribute
        (Globals.MBEAN_REGISTRY_ATTR, MBeanUtils.createRegistry());
    context.getServletContext().setAttribute
        (Globals.MBEAN_SERVER_ATTR, MBeanUtils.createServer());
}
...
}

```

如果 Context 实例的 privileged 属性为 true，则会为 Web 应用程序添加两个属性，并将其存储在 ServletContext 对象中。两个属性的键名分别是 Globals.MBEAN\_REGISTRY\_ATTR 和 Globals.MBEAN\_SERVER\_ATTR。下面是 org.apache.catalina.Globals 类的代码片段：

```

/**
 * The servlet context attribute under which the managed bean Registry
 * will be stored for privileged contexts (if enabled).
 */
public static final String MBEAN_REGISTRY_ATTR =
    "org.apache.catalina.Registry";

/**
 * The servlet context attribute under which the MBeanServer will be
 * stored for privileged contexts (if enabled).
 */
public static final String MBEAN_SERVER_ATTR =
    "org.apache.catalina.MBeanServer";

```

MBeanUtils 类的 createRegistry() 方法返回一个 Registry 实例，而 createServer() 方法返回一个 javax.management.MBeanServer 实例。Catalina 中所有的 MBean 都注册于 MBeanServer 实例中。

换句话说，当 privileged 属性为 true 时，才可以从一个 Web 应用程序中获取 Registry 类和 MBeanServer 类的对象。下一节将会讨论如何创建 JMX 管理器应用程序来管理 Tomcat。

## 20.8 应用程序

该应用程序是用来管理 Tomcat 的 Web 应用程序。虽然它很简单，但已足够说明如何使用 Catalina 暴露出的 MBean。可以列出 Catalina 中所有的 ObjectName 实例，以及当前正在运行的所有 Context 实例，并删除其中的任意一个。

首先要为该 Web 应用程序创建描述符文件，如代码清单 20-18 所示，该描述文件必须要放到 %CATALINA\_HOME%/webapps 目录下：

代码清单 20-18 myadmin.xml 文件

```

<Context path="/myadmin" docBase="../server/webapps/myadmin" debug="8"
privileged="true" reloadable="true">
</Context>

```

其中需要注意的是，Context 元素的 privileged 属性的值必须为 true。docBase 属性指定了



Web 应用程序的文件路径。

该 Web 应用程序包含一个 servlet，如代码清单 20-19 所示。

代码清单 20-19 MyAdminServlet 类

```
package myadmin;

import java.io.IOException;
import java.io.PrintWriter;
import java.net.URLEncoder;
import java.util.Iterator;
import java.util.Set;
import javax.management.MBeanServer;
import javax.management.ObjectName;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.commons.modeler.Registry;

public class MyAdminServlet extends HttpServlet {
    private Registry registry;
    private MBeanServer mBeanServer;

    public void init() throws ServletException {
        registry = (Registry)
            getServletContext().getAttribute("org.apache.catalina.Registry");
        if (registry == null) {
            System.out.println("Registry not available");
            return;
        }
        mBeanServer = (MBeanServer) getServletContext().getAttribute(
            "org.apache.catalina.MBeanServer");
        if (mBeanServer == null) {
            System.out.println("MBeanServer not available");
            return;
        }
    }

    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        if (registry == null || mBeanServer == null) {
            out.println("Registry or MBeanServer not found");
            return;
        }
        out.println("<html><head><body>");
        String action = request.getParameter("action");
        if ("listAllManagedBeans".equals(action)) {
            listAllManagedBeans(out);
        }
        else if ("listAllContexts".equals(action)) {
            listAllContexts(out);
        }
        else if ("removeContext".equals(action)) {
            String contextObjectName =
                request.getParameter("contextObjectName");
            // ... (rest of the code) ...
        }
    }
}
```

```

        removeContext(contextObjectName, out);
    }
    else {
        out.println("Invalid command");
    }
    out.println("</body></html>");
}

private void listAllManagedBeans(PrintWriter out) {
    String[] managedBeanNames = registry.findManagedBeans();
    for (int i=0; i<managedBeanNames.length; i++) {
        out.print(managedBeanNames[i] + "<br/>");
    }
}

private void listAllContexts(PrintWriter out) {
    try {
        ObjectName objName = new ObjectName("Catalina:type=Context,*");
        Set set = mBeanServer.queryNames(objName, null);
        Iterator it = set.iterator();
        while (it.hasNext()) {
            ObjectName obj = (ObjectName) it.next();
            out.print(obj +
                " <a href=?action=removeContext&contextObjectName=" +
                URLEncoder.encode(obj.toString(), "UTF-8") +
                ">remove</a><br/>");
        }
    }
    catch (Exception e) {
        out.print(e.toString());
    }
}

private void removeContext(String contextObjectName,
    PrintWriter out) {
    try {
        ObjectName mBeanFactoryObjectName = new
            ObjectName("Catalina:type=MBeanFactory");
        if (mBeanFactoryObjectName!=null) {
            String operation = "removeContext";
            String[] params = new String[1];
            params[0] = contextObjectName;
            String signature[] = { "java.lang.String" };
            try {
                mBeanServer.invoke(mBeanFactoryObjectName, operation,
                    params, signature);
                out.println("context removed");
            }
            catch (Exception e) {
                out.print(e.toString());
            }
        }
    }
    catch (Exception e) {
    }
}
}

```

最后，需要一个应用程序部署描述符文件。该文件的内容在代码清单 20-20 中给出。

代码清单 20-20 web.xml 文件

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>myAdmin</servlet-name>
    <servlet-class>myadmin.MyAdminServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>myAdmin</servlet-name>
    <url-pattern>/myAdmin</url-pattern>
  </servlet-mapping>
</web-app>
```

若想要列出所有的 ObjectName 实例，可以使用下面的 URL 地址：

<http://localhost:8080/myadmin/myAdmin?action=listAllMBeans>

你将会看到 MBean 对象的列表，下面是输出结果的一部分：

```
MemoryUserDatabase
DigestAuthenticator
BasicAuthenticator
UserDatabaseRealm
SystemErrLogger
Group
```

要想列出所有正在运行的 Web 应用程序，可以使用如下的 URL：

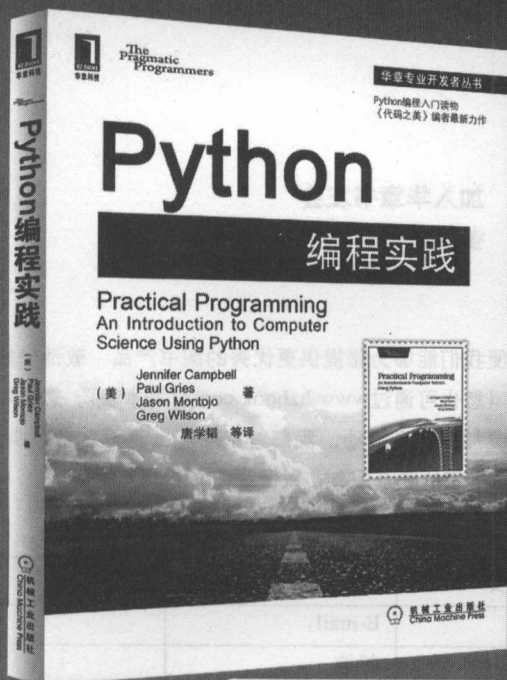
<http://localhost:8080/myadmin/myAdmin?action=listAllContexts>

你将会看到所有正在运行的 Web 应用程序。若想要移除其中某个 Web 应用程序，可以单击“remove”超链接。

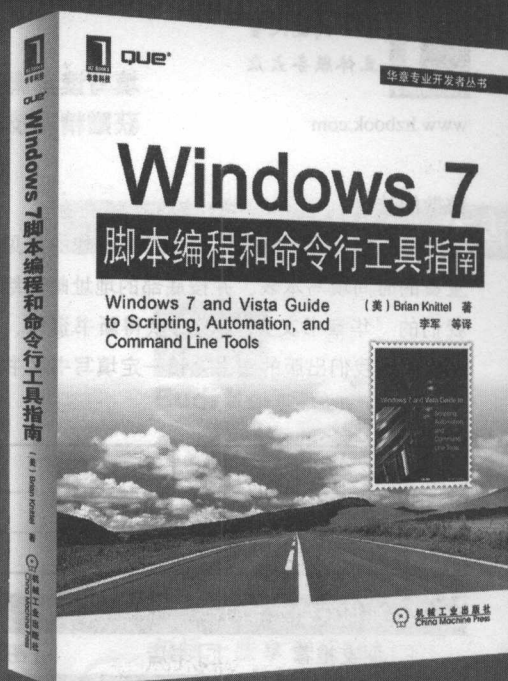
## 20.9 小结

在本章中，你已经学会了如何使用 JMX 管理 Tomcat，并学习了 4 种 MBean 类型中标准 MBean 和模型 MBean。最后，使用 Catalina 提供的 MBean 开发了一个简单的 Admin 应用程序用来管理 Tomcat。

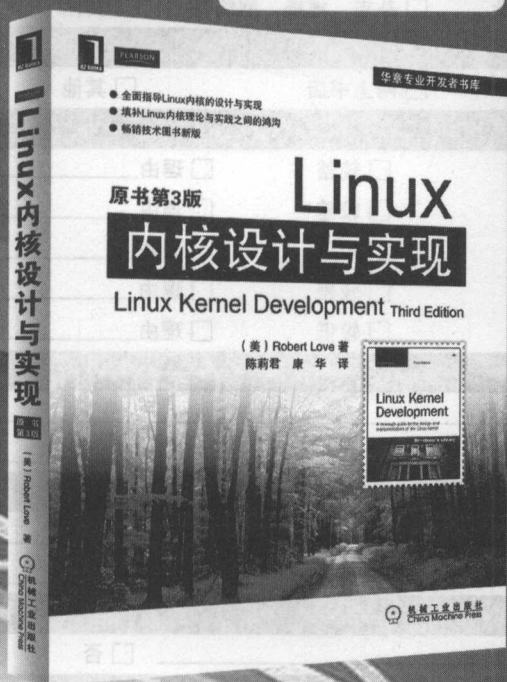




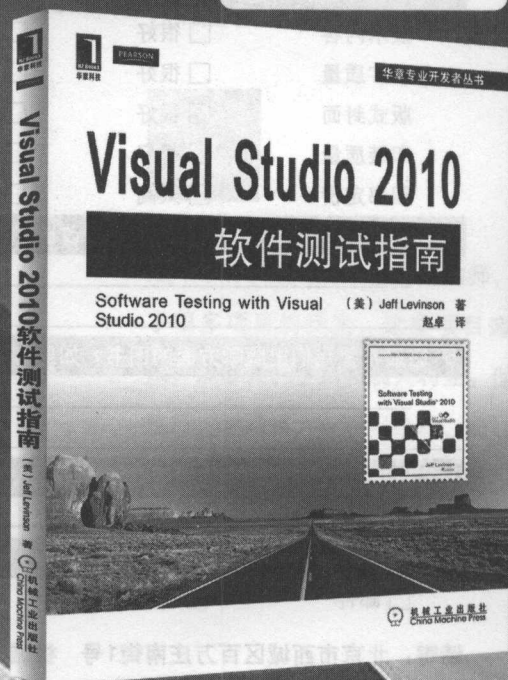
ISBN 978-7-111-36478-8  
定价: 49.00元



ISBN 978-7-111-35677-6  
定价: 79.00元



ISBN 978-7-111-33829-1  
定价: 69.00元



ISBN 978-7-111-35931-9  
定价: 49.00元



机械工业出版社  
China Machine Press

## 作者简介



### **Budi Kurniawan**

是一位IT咨询师，主要研究方向包括互联网与面向对象程序设计。除了其他一些计算机图书，他已经在10多种出版物上发表了约100篇文章，其中包括一些享有盛誉的Java杂志，如《JavaPro》、《JavaWorld》等。Budi也是Brainysoftware.com的应用程序File Upload Bean的作者。



### **Paul Deck**

是一个有15多年经验的IT架构师，参与了很多项目的开发，这些项目遍布美国、加拿大、中国和澳大利亚。他喜欢旅行，对网络、互联网编程、设计模式和用户界面交互设计非常感兴趣。

# 深入剖析 Tomcat

## How Tomcat Works A Guide to Developing Your Own Java Servlet Container

本书深入剖析Tomcat 4和Tomcat 5中的每个组件，并揭示其内部工作原理。通过学习本书，你不仅可以自行开发Tomcat组件，而且可以扩展已有的组件。

### 本书主要内容：

- 如何开发Java Web服务器。
- Tomcat是否会为每个servlet类创建多个实例。
- Tomcat如何运行一个实现SimpleThreadModel接口的servlet类。
- servlet容器的两个主要模块：连接器和servlet容器。
- 如何构建或者扩展已有的连接器。
- 4种servlet容器：Engine、Host、Context和Wrapper。
- Tomcat如何管理Session，以及如何在分布式环境下扩展Session管理器。
- Tomcat中的类加载器和如何创建自定义加载器。
- Tomcat如何实现安全性和基本/基于表单/摘要的身份验证。
- Tomcat中的领域与登录配置是如何工作的。
- Tomcat如何处理配制文件（server.xml），以及如何通过Digester库将XML元素转换为Java对象。
- Tomcat中的关闭钩子。
- JMX、Apache的 Commons Modeler和Tomcat中的JMX托管资源。

客服热线：(010) 88378991, 88361066  
购书热线：(010) 68326294, 88379649, 68995259  
投稿热线：(010) 88379604  
读者信箱：hzsj@hzbook.com

华章网站 <http://www.hzbook.com>

网上购书：[www.china-pub.com](http://www.china-pub.com)

封面设计 范华明



上架指导：计算机/程序设计

ISBN 978-7-111-36997-4



9 787111 369974

定价：59.00元